#123 January 1987

2.95 (3.95 CANADA)

Dr. Dobb's Journal of Software Tools

FOR THE PROFESSIONAL PROGRAMMER

ANNUAL 68K ISSUE

68K Mini Forth

OS-9 Operating System

Mac and Amiga Interface Programming

Languages:
- Forth Names
- Proper PROLOG
- Memory Management in C
- BASIC Rebirth
- 68K Assembly Project

The Bandwidth Bottleneck

0 38351 16562 8

Instant Replay

Has It Dawned On You Yet?
Prototype-->Program-->Tutorial

) **Instant Replay - $ 89.95**
Demo, Tutorial, and Prototype Generator. Create Replays of actual programs, or vapor ware only replays. Includes Screen Genie Screen Editor, Word Genie Text Editor, and a Keystroke Editor. Insert Pop-Ups, Prompts, Prototypes, and Users into the demo or tutorial.

) **Assembler Genie $ 59.95**
Assembly Language source code generator for including Screen Genie prototype screens into your actual programs. Supports almost every language.

) **Plus Series - $ 59.95**
Assembly Language Programming Tools for Turbo Pascal, MS Pascal, or C. Make your prototypes into real programs.

) **NoBlink Accelerator - $ 49.95**
Resident Cursor Enhancer. Choose your own cursor shape, color(non-blinking), or speed in any program. Uses only 6K of memory. Great time time saver. Why mouse around?

60 day satisfaction money back guarantee on all software. Not Copy Protected.

o order, or to obtain more information call or write
ostradamus Inc. (801) 487-9662
91 South Valley Street (suite 252)
t Lake City, Utah 84109

NOSTRADAMUS
Software Sourcery
Circle **no. 251** on reader service card.

Amex, Master, Cod, PO, Check
M PC, XT, AT and True Compatibles.
e US pays postage.

JANUARY 1987 VOLUME 12, ISSUE 1

CONTENTS

ARTICLES

The 68K story ▶

Mac and Amiga assembly language ▶

680xx PROGRAMMING: 680xx Computers: Where Are They Going? 16
by Nick Turner
An overview of the 680xx family of chips: past, present, and probable future.

680xx PROGRAMMING: A Mini Forth for the 68000 22
by G. Yates Fletcher
Yates tells us about the "no frills" Forth-like interpreter for the 68000 that he designed to test the theory that Forth is more naturally understood as a program than as a language.

680xx PROGRAMMING: The OS-9 Operating System 30
by Brian Capouch
A look at the modular, multiprogramming, multitasking operating system growing in popularity among 680xx programmers.

680xx PROGRAMMING: Macintosh Buttons and Amiga Gadgets 40
by Jan L. Harrington
Comparing user interface programming on the Macintosh and Amiga, Jan provides details about the operating system support produced on both machines for user interface features such as menus, buttons and windows.

PROCESSORS: Series 32000 Cross Assembler 82
by Richard Rodman
The listing (in human-readable form) for Richard's article that was published in December.

COLUMNS

Managing your memory ▶

Forth names ▶

C CHEST 104
by Allen Holub
Allen presents some memory management techniques plus an explanation of C memory organization for beginning C programmers.

STRUCTURED PROGRAMMING 110
by Michael Ham
Michael discusses the naming of names in Forth.

THE RIGHT TO ASSEMBLE 126
by Nick Turner
Nick launches a project to design a versatile, easy-to-use, interpreted language to be written in 680xx assembly language.

FORUM

The bandwidth bottleneck

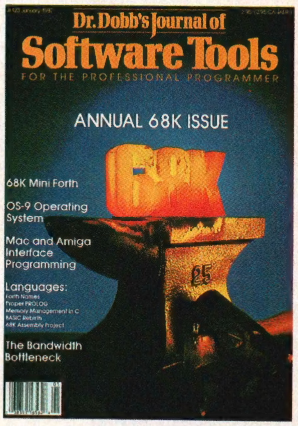
Proper PROLOG ▶

The rebirth of BASIC ▶

EDITORIAL 6
by Michael Swaine
RUNNING LIGHT 8
by Nick Turner
ARCHIVES 8
LETTERS 10
by you
VIEWPOINT 14
by Dick Butrick
DDJ ON LINE 130
SWAINE'S FLAMES 152
by Michael Swaine

PROGRAMMER'S SERVICES

DR. DOBB'S CATALOG: 117
DDJ books and software
OF INTEREST: 140
New products out there
ADVERTISER INDEX: 151
Where to find those ads



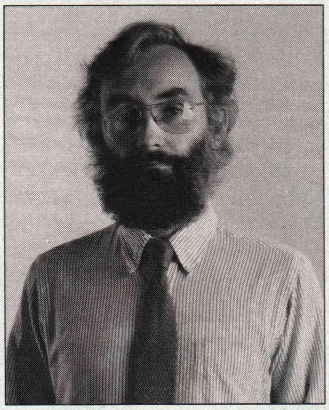
About the Cover
Motorola has just forged the 68030. Is it as hot as it seems?

This Issue
In the beginning there were the 8080 and the 6502—programmers chose their weapons and the battle lines were drawn. A few years later, Motorola gave the "sixers" more power when it introduced its 680xx line of chips. Today there is a wide range of powerful 680xx machines—and some very interesting rumors about the future. This month we survey the 680xx family and examine a modular, multitasking operating system, a 68K Forth-like interpreter, and the challenges of creating Amiga- and Mac-like user interfaces.

Next Issue
The choice of a text editor is based on many highly subjective considerations as well as some "hard" pragmatic requirements. In February, we'll present an overview of the various elements involved in that choice and let you hear what some programmers have to say about their favorite and least favorite editors.

—bandwidth topic
—entry point

EDITORIAL



Here's some of what we have planned for *DDJ*'s twelfth year.

In a series of short reports, contributing editor Namir Shammas will examine the state of the BASIC language in 1987. Yes, BASIC. *DDJ* is hardly a beginner's magazine, but if the beginner's language has grown up to long pants, as some claim, we should all know about it.

Then again, each of us is a novice in some area. Realizing this, *DDJ* will use the icon at the beginning of this paragraph to flag certain features as entry points: items such as Allen Holub's Flotsam and Jetsam in this issue, from which the less experienced programmer can learn useful techniques or gain familiarity with more technical subjects.

There are a lot of ways to write about artificial intelligence, nearly all bad. Our new column on alternative programming paradigms (it starts next month) will avoid them all as it examines such topics as knowledge-based programming, logic programming, and object-oriented programming from an experienced software designer's perspective. Contributing editor Ernie Tello writes, lectures, and consults in this area and promises to take us beyond the fields we know.

And we're going to attack the bandwidth problem.

DDJ was born to shoehorn BASIC into the hypomnemonic personal computers of 1976. In a sense, the Cain/Hendrix versions of Small-C published in *DDJ* over the succeeding years addressed this same problem of cramming programming power into micro memory. You could say that's been the charter of the magazine, and on one level it will continue to be our focus. But nobody today needs another Tiny BASIC or Small-C. Developments like the Intel 80386 processor lift the lid of a different box of programming problems. One is the efficient transmission of information over limited channels.

Bandwidth is already an issue in graphics output: Microsoft Windows never made sense on 8088 machines and the PARC interface overwhelmed the original 128K Mac. Adequate memory and processor power make a big difference, but they may ultimately just move the bottleneck to the IC level.

Bandwidth becomes more of an issue in mass storage as storage becomes more massive. Once we learn what to do with CD-ROMs, retrieving information efficiently from them will require more than increased speed of transmission.

The potential bandwidth crunch in telecommunications and remote database access is obvious, but when LANs start proliferating, so will in-house bandwidth competition.

Approaches to the bandwidth crunch can range from clever data-compression algorithms to systems that form hypotheses about incoming data and acquire just enough data to confirm or reject the hypotheses. As access to information becomes more technically problematic, it will also take on sociopolitical dimensions; for example, public libraries are becoming measurably less public as they subscribe to commercial information-provider services and pass the costs on to their patrons. We'll delve into the technology for dealing with the bandwidth crunch while trying to see its potential social consequences.

Bandwidth-related items will be flagged with the icon at the end of this line.

Michael Swaine

Michael Swaine
editor-in-chief

Dr. Dobb's Journal of Software Tools

Editorial
Editor-in-Chief *Michael Swaine*
Editor *Nick Turner*
Managing Editor *Vince Leone*
Assistant Editors *Sara Noah Ruddy*, *Levi Thomas*
Technical Editor *Allen Holub*
Contributing Editors *Ray Duncan*, *Michael Ham*, *Bela Lubkin*, *Namir Shammas*, *Ernest R. Tello*
Copy Editor *Rhoda Simmons*

Production
Production Manager *Bob Wynne*
Art Director *Michael Hollister*
Assoc. Art Director *Joe Sikoryak*
Typesetter *Jean Aring*
Cover Artist *Michael Carr*

Circulation
Circulation Director *Maureen Kaminski*
Newsstand Sales Mgr. *Stephanie Barber*
Book Marketing Mgr. *Jane Sharninghouse*
Circulation Coordinator *Kathleen Shay*

Administration
Finance Director *Kate Wheat*
Business Manager *Betty Trickett*
Accounts Payable Supv. *Mayda Lopez-Quintana*
Accounts Receivable Mgr. *Laura Di Lazzaro*

Advertising Director
Robert Horton *(415) 366-3600*

Account Managers
Michele Beaty *(317) 875-8093*
Lisa Boudreau *(415) 366-3600*
Gary George *(404) 897-1923*
Michael Wiener *(415) 366-3600*
Cynthia Zuck *(718) 499-9333*
Promotions/Srvcs.Mgr. *Anna Kittleson*
Advertising Coordinator *Charles Shively*

M&T Publishing Inc.
Chairman of the Board *Otmar Weber*
Director *C.F. von Quadt*
President and Publisher *Laird Foshay*

Dr. Dobb's Journal of Software Tools (USPS 307690) is published monthly by M&T Publishing Inc., 501 Galveston Dr., Redwood City, CA 94063; (415) 366-3600. Second-class postage paid at Redwood City and at additional entry points.

Article Submissions: Send manuscripts and disk (with article and listings) to the Editor.

***DDJ* on CompuServe:** Type GO DDJ

Address Correction Requested: Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, P.O. Box 27809, San Diego, CA 92128. **ISSN 0888-3076**

Customer Service: For subscription problems call: outside CA (800) 321-3333; in CA (619) 485-9623 or 566-6947. For book/software order problems call (415) 366-3600.

Subscriptions: $29.97 per year; $56.97 for 2 years. Canada and Mexico add $27 per year airmail or $10 per year surface. All other countries add $45 per year airmail or $28 per year surface. Foreign subscriptions must be prepaid in U.S. funds drawn on a U.S. bank. For foreign subscriptions, TELEX: 752-351.

Foreign Newsstand Distributor: Worldwide Media Service Inc., 386 Park Ave. South, New York, NY 10016; (212) 686-1520 TELEX: 620430 (WUI).

People's Computer Company

Dr. Dobb's Journal of Software Tools is published by M&T Publishing Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a nonprofit corporation.

AZTEC C — The C for Microcomputers

PC-DOS, MS-DOS, CP/M-86, Macintosh, Amiga, Apple II, CP/M-80, Radio Shack, Commodore, XENIX, ROM, and Cross Development systems

MS-DOS, PC-DOS, CP/M-86, XENIX, 8086/80x86 ROM

Manx Aztec C86

"A compiler that has many strengths . . . quite valuable for serious work"

Computer Language review, February 1985

Great Code: Manx Aztec C86 generates fast executing compact code. The benchmark results below are from a study conducted by Manx. The Dhrystone benchmark (CACM 10/84 27:10 p1018) measures performance for a systems software instruction mix. The results are without register variables. With register variables, Manx, Microsoft, and Mark Williams run proportionately faster, Lattice and Computer Innovations show no improvement.

	Execution Time	Code Size	Compile/ Link Time
Dhrystone Benchmark			
Manx Aztec C86 3.3	34 secs	5,760	93 secs
Microsoft C 3.0	34 secs	7,146	119 secs
Optimized C86 2.20J	53 secs	11,009	172 secs
Mark Williams 2.0	56 secs	12,980	113 secs
Lattice 2.14	89 secs	20,404	117 secs

Great Features: Manx Aztec C86 is bundled with a powerful array of well documented productivity tools, library routines and features.

Optimized C compiler	Symbolic Debugger
AS86 Macro Assembler	LN86 Overlay Linker
80186/80286 Support	Librarian
8087/80287 Sensing Lib	Profiler
Extensive UNIX Library	DOS, Screen, & Graphics Lib
Large Memory Model	Intel Object Option
Z (vi) Source Editor -c	CP/M-86 Library -c
ROM Support Package -c	INTEL HEX Utility -c
Library Source Code -c	Mixed memory models -c
MAKE, DIFF, and GREP -c	Source Debugger -c
One year of updates -c	CP/M-86 Library -c

Manx offers two commercial development systems, Aztec C86-c and Aztec C86-d. Items marked -c are special features of the Aztec C86-c system.

Aztec C86-c Commercial System	**$499**
Aztec C86-d Developer's System	**$299**
Aztec C86-p Personal System	**$199**
Aztec C86-a Apprentice System	**$49**

All systems are upgradable by paying the difference in price plus $10.

Third Party Software: There are a number of high quality support packages for Manx Aztec C86 for screen management, graphics, database management, and software development.

C-tree $395	**Greenleaf $185**
PHACT $250	**PC-lint $98**
HALO $250	**Amber Windows $59**
PRE-C $395	**Windows for C $195**
WindScreen $149	**FirsTime $295**
SunScreen $99	**C Util Lib $185**
PANEL $295	**Plink-86 $395**

MACINTOSH, AMIGA, XENIX, CP/M-68K, 68k ROM

Manx Aztec C68k

"Library handling is very flexible . . . documentation is excellent . . . the shell a pleasure to work in . . . blows away the competition for pure compile speed . . . an excellent effort."

Computer Language review, April 1985

Aztec C68k is the most widely used commercial C compiler for the Macintosh. Its quality, performance, and completeness place Manx Aztec C68k in a position beyond comparison. It is available in several upgradable versions.

Optimized C	Creates Clickable Applications
Macro Assembler	Mouse Enhanced SHELL
Overlay Linker	Easy Access to Mac Toolbox
Resource Compiler	UNIX Library Functions
Debuggers	Terminal Emulator (Source)
Librarian	Clear Detailed Documentation
Source Editor	C-Stuff Library
MacRam Disk -c	UniTools (vi,make,diff,grep) -c
Library Source -c	One Year of Updates -c

Items marked -c are available only in the Manx Aztec C86-c system. Other features are in both the Aztec C86-d and Aztec C86-c systems.

Aztec C68k-c Commercial System	**$499**
Aztec C68d-d Developer's System	**$299**
Aztec C68k-p Personal System	**$199**
C-tree database (source)	**$399**
AMIGA, CP/M-68k, 68k UNIX	**call**

Apple II, Commodore, 65xx, 65C02 ROM

Manx Aztec C65

"The AZTEC C system is one of the finest software packages I have seen"

NIBBLE review, July 1984

A vast amount of business, consumer, and educational software is implemented in Manx Aztec C65. The quality and comprehensiveness of this system is competitive with 16 bit C systems. The system includes a full optimized C compiler, 6502 assembler, linkage editor, UNIX library, screen and graphics libraries, shell, and much more. The Apple II version runs under DOS 3.3, and ProDOS, Cross versions are available.

The Aztec C65-c/128 Commodore system runs under the C128 CP/M environment and generates programs for the C64, C128, and CP/M environments. Call for prices and availability of Apprentice, Personal and Developer versions for the Commodore 64 and 128 machines.

Aztec C65-c ProDOS & DOS 3.3	**$399**
Aztec C65-d Apple DOS 3.3	**$199**
Aztec C65-p Apple Personal system	**$99**
Aztec C65-a for learning C	**$49**
Aztec C65-c/128 C64, C128, CP/M	**$399**

Distribution of Manx Aztec C

In the USA, Manx Software Systems is the sole and exclusive distributor of Aztec C. Any telephone or mail order sales other than through Manx are unauthorized.

Manx Cross Development Systems

Cross developed programs are edited, compiled, assembled, and linked on one machine (the HOST) and transferred to another machine (the TARGET) for execution. This method is useful where the target machine is slower or more limited than the HOST, Manx cross compilers are used heavily to develop software for business, consumer, scientific, industrial, research, and educational applications.

HOSTS: VAX UNIX ($3000), PDP-11 UNIX ($2000), MS-DOS ($750), CP/M ($750), MACINTOSH ($750), CP/M-68k ($750), XENIX ($750).

TARGETS: MS-DOS, CP/M-86, Macintosh, CP/M-68k, CP/M-80, TRS-80 3 & 4, Apple II, Commodore C64, 8086/80x86 ROM, 68xxx ROM, 8080/8085/Z80 ROM, 65xx ROM.

The first TARGET is included in the price of the HOST system. Additional TARGETS are $300 to $500 (non VAX) or $1000 (VAX).

Call Manx for information on cross development to the 68000, 65816, Amiga, C128, CP/M-68K, VRTX, and others.

CP/M, Radio Shack, 8080/8085/Z80 ROM

Manx Aztec CII

"I've had a lot of experience with different C compilers, but the Aztec C80 Compiler and Professional Development System is the best I've seen."

80-Micro, December, 1984, John B. Harrell III

Aztec C II-c (CP/M & ROM)	**$349**
Aztec C II-d (CP/M)	**$199**
C-tree database (source)	**$399**
Aztec C80-c (TRS-80 3 & 4)	**$299**
Aztec C80-d (TRS-80 3 & 4)	**$199**

How To Become an Aztec C User

To become an Aztec C user call 1-800-221-0440 or call 1-800-832-9273 (800-TEC WARE). In NJ or outside the USA call 201-530-7997. Orders can also be telexed to 4995812.

Payment can be by check, COD, American Express, VISA, Master Card, or Net 30 to qualified customers.

Orders can also be mailed to Manx Software Systems, Box 55, Shrewsbury, NJ 07701.

How To Get More Information

To get more information on Manx Aztec C and related products, call 1-800-221-0440, or 201-530-7997, or write to Manx Software Systems.

30 Day Guarantee

Any Manx Aztec C development system can be returned within 30 days for a refund if it fails to meet your needs. The only restrictions are that the original purchase must be directly from Manx, shipped within the USA, and the package must be in resalable condition. Returned items must be received by Manx within 30 days. A small restocking fee may be required.

Discounts

There are special discounts available to professors, students, and consultants. A discount is also available on a "trade in" basis for users of competing systems. Call for information.

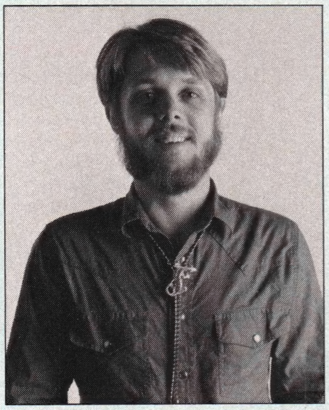
MANX

To order or for information call:

800-221-0440

Circle **no. 108** on reader service card.

RUNNING LIGHT



What is the logical end point of the current trend toward smaller and smaller chip components? How soon will it be reached, and how reliable will the components be at that level? These questions are being addressed by a totally new branch of science, one that combines the disciplines of chemistry, physics, biology, computer science, and mathematics. The new field, called nanotechnology, concerns itself with devices that are several orders of magnitude smaller than current microcircuitry—components made up of individual atoms and molecules carefully bonded one by one. This may seem too farfetched to expect to see it in our lifetimes, but we're much, much closer than that. Several recent developments have suddenly elevated nanotechnology from a pipe dream to a bona fide discipline. I recently attended a seminar at which the field's most vocal advocate, Eric Drexler, spoke eloquently about nanotechnology. He actually laid out the design (on an atom-by-atom scale) of a working nanocomputer, smaller than a virus particle, along with all its support devices. I'm excited by the potential of such incredibly small, fast devices, and I'd love to hear from any of you who are working in this new field.

We've received excellent responses to our October 1986 issue, which focused on the 80386 and its family. This month we focus on the other side, the sixers. (If you're a 680xx programmer, you're probably a sixer.) We start with an overview of the 68000 line—where it has been, where it is now, and where it might be going. As more information becomes available on the 68040 (and beyond), we'll keep you up to date.

Yates Fletcher has written a Forth-like interpreter for the 68000 that he calls FLINT. His article in this issue describes it in detail and offers some interesting insight into what makes threaded interpreters so efficient. Also in this issue, Brian Capouch takes a look at the OS-9 operating system from Microware. OS-9 shows promise as a standard for 68K development work, and Brian explains why. Jan Harrington's article about Amiga gadgets and Macintosh buttons is one of the clearest comparisons I've seen of the different programming styles required on the two machines.

This month I begin what I hope will be a series of essays on the design of an interpreted language for the 68000. I'm interested in your comments and criticism.

Last July, in our annual Forth issue, we published an article about a Forth-driven robot that dives into the ocean, records oceanic data, and then pops up to send the data home via satellite. This July we'd like to focus on embedded systems of that sort—programs that reside inside autonomous devices. If you're working on such a device, we'd like to hear from you.

Bela Lubkin has joined Levi Thomas and Ray Duncan as a sysop on our CompuServe SIG (DDJFORUM) and has been stirring things up quite nicely. You will doubtless see his name on many a message in DDJ On Line.

Finally, I'd like to thank Jerry Houston, Roger Dunn, John Berry, Charles Marslett, and Wayne Vucenic for their help with Table 2 on page 18.

Nick Turner
editor

ARCHIVES

The 68000 et Famille

[A small boy, Oliver Wendall Jones, is sitting on Santa's lap reciting his Christmas wish list.]

Oliver: "That's the 32-bit MC68000 microprocessor . . . not to be confused with. . . ."

Santa: "The 16-bit 8088. Total garbage, El Stinko."—*Bloom County (comic strip), Berke Breathed, December 12, 1985.*

"Tom Pittman, the implementor of the $5/copy version of Tiny BASIC for the 6800 and 650x is now offering an experimenter's kit for those folks who want to modify and extend his Tiny BASIC. The kit includes an assembled source listing for the IL (Intermediate Language), an IL assembler written in Tiny BASIC, a detailed description of the virtual machine implemented by the IL, instructions for incorporating a new IL into the Tiny BASIC system, and finally some practical hints about debugging and extending the system. The cost is $10 from Itty Bitty Computers."—DDJ, *March 1977.*

Flak from Our Readers

"A language design frequency of almost one Tiny BASIC version per month is certainly impressive. Programmers ought to have realized that God invented many different laguages in Babylon not to enjoy but to punish mankind, and, after all, he never intended implementing all of them for the 8080. So why not write—and, if necessary, publish—programs in an informal, ad hoc language fitting to the problem, not the computer?"—*Thomas Alexander Matzner, letter to the editor,* DDJ, *March 1977.*

"[You write 'em; we'll publish (maybe).]"—*editorial reply to the above,* DDJ, *March 1977.*

Ten Years Ago in *DDJ*

"Why bother with a multi-tasking operating system on a 'personal' computer? Let's daydream for a moment. Wouldn't it be nice to be able to start a lengthy listing on our hardcopy device; while that was running, start an assembly of a large program; and then go about editing the source for another program from our softcopy terminal? That's EXACTLY what you can do with a multi-tasking system."—*Jim Warren,*DDJ, *January 1977.*

"I would like to particularly applaud Dick [Wilcox]'s position regarding low-cost distribution of software for not-for-profit use. He is recognizing and adjusting to the realities of the new world of personal computing in a manner that I feel is fair and reasonable for everyone concerned."—*Jim Warren,* DDJ, *January 1977.*

DR. DOBB'S JOURNAL of COMPUTER Calisthenics & Orthodontia
Running Light Without Overbyte

Large model C compiler performs like big-name brands includes free C tutorial

$99

"As good as or better than most of the heavyweights..."
DR. DOBB'S, August 1986

The DATALIGHT C COMPILER is a full-implementation of the C language as defined in *The C Programming Language* by Kernighan and Ritchie. Supporting five memory models, DATALIGHT C has very fast compile, link, and execution times with a minimum of memory required. Our special introductory price is only $99 for the DEVELOPER'S KIT.

Optimize Your Code Generation

Now you can produce highly optimized code in the standard Intel object module format. The optimizations performed include common subexpression elimination, branch optimizations, constant folding, strength reductions, dead-code eliminations, and switch table compaction.

Five Memory Models Supported

DATALIGHT C provides five memory models so you can use the model that best suits your application.

Memory Models

Model	Code	Data
Compact	64k	total code & data
Small	64k	64k
Program	1M	64k
Data	64k	1M
Large	1M	1M

Compiling, One step...

Now with the one step DLC program you can compile, link, and create a .COM file. Also, one or more files can be compiled and linked using DLC.

Complete Library Includes Source Code

The UNIX compatible library includes complete source code. Experienced programmers can use the source code to configure and rebuild the library to suit the application.

Concise Documentation Included

The DATALIGHT C COMPILER and DEVELOPER'S KIT include a concise, to the point, programmer's manual. The 210-page manual is contained in an IBM-style three-ring binder which includes nine chapters, appendices, index, and easy to follow examples.

Datalight

BOX 82441
KENMORE, WA 98028
(206) 367-1803

30-DAY MONEY-BACK GUARANTEE

Try our DEVELOPER'S KIT for 30 days, and if you do not find it to be equal to, or better than, any compiler you have, or are using, then you may return it for a full refund. Also, if you return the compiler within 30 days, you may keep the C tutorial, a $39 value, for trying our DEVELOPER'S KIT.

Introductory Prices

DATALIGHT C without source $60
(compact & small memory model support)

DEVELOPER'S KIT with source $99
(five memory models supported)

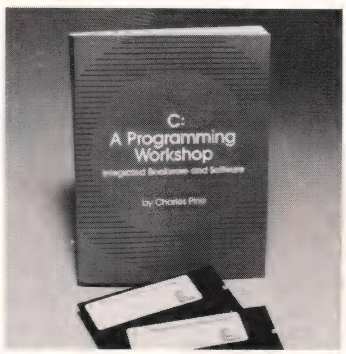
Not Copy Protected

ORDER TOLL-FREE TODAY!

1-(800) 221-6630

*SPECIAL OFFER!

ORDER the DEVELOPER'S KIT by January 31, 1987 and receive free....



"Combines print and software technology to create the most integrated new type of training system I have ever seen."

ADAM GREEN, INFO WORLD
January 27, 1986

"C:A Programming Workshop" is a complete C language learning package with an integrated compiler and screen editor. Learn the C language with tutorials, quizzes, and program exercises.

Add $5 for shipping in US/$15 outside US
COD (add $2.50)

*Limited offer available exclusively to readers who purchase directly from DATALIGHT.

Magazine Reviewers Shocked by DATALIGHT'S PERFORMANCE...

"Reviewing this compiler was quite a surprise for us. For such a low price, we were expecting a "lightweight" compiler. What we got was a package that is as good as or better than most of the "heavyweights." DATALIGHT C implements a complete C language. It also compiles quickly, doesn't take up much disk space, and looks impressive in the benchmarks."

DR. DOBB'S, August 1986

"This is a sharp compiler! ...what is impressive is that DATALIGHT not only stole the compile time show completely, but had the fastest Fibonacci executable time and had excellent object file sizes to boot!"

Chris Skelly, COMPUTER LANGUAGE
February 1986

DEVELOPER'S KIT (VERSION 2.12)

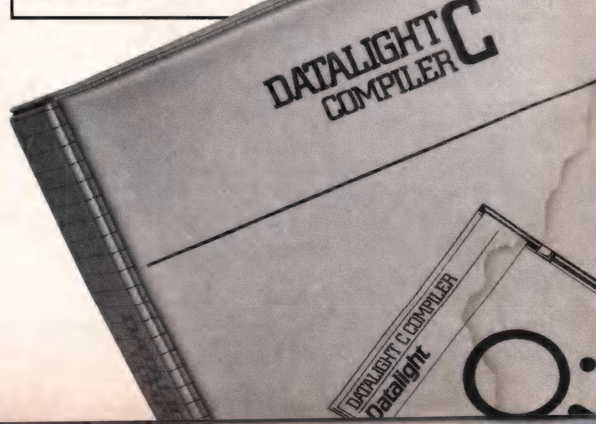
- Full UNIX System 5 C language plus ANSI extensions.
- Fast/tight code via powerful optimizations including common sub-expression elimination.
- DLC one step compile/link program.
- Multiple memory model support.
- UNIX compatible library with PC functions.
- Compatible with DOS linker and assembler.
- Third-party library support.
- Automatic generation of .COM files.
- Supports DOS pathnames, wild cards, and Input/Output redirection.
- Compatible with Lattice C version 2.x.
- Interrupt handling in C.
- Debugger support.
- ROMable code support/start-up source.

MAKE Maintenance Utility

- Macro definition support.
- MS-DOS internal commands.
- Inference rule support.
- TOUCH date manager.

Tools in Source Code

- cat—UNIX style "type"
- diff—Text file differences
- fgrep—Fast text search
- pr—Page printer
- pwd—Print working directory
- wc—Word count



Circle no. 203 on reader service card.

LETTERS

Software Gap

Dear *DDJ*,

After reading Nick Turner's Running Light column in the August issue concerning the perceived "software gap," I felt as though I had to tell you how I see it.

I hold a Bachelor of Science degree in computer science from the University of Maryland. I learned all about the "right" way to design and code programs, including everything you say programmers don't care about anymore. From my own experience, I find this to have been a waste of time and tuition.

The biggest problem I have had in my "career" has been convincing some of the "data processing managers" how a program should be constructed. Every time I try to write some form of documentation, I am told "not to waste time on such useless paperwork." I wish I could say this only happened at one or two companies, but I have been employed at four different companies over the past five years, and every one of them has been the same. I am beginning to think the only way I'll ever get to be what you call a "professional" programmer is to start my own software company.

At times I wonder if the problem with the management structure stems from the fact that most of the people promoted have business backgrounds. Not one of my bosses has ever had anything other than a business degree, and not one knows the first thing about a programming project. My current boss only understands straight-line coding and sequential list processing (that is, no doubly linked lists, no sparse matrices, no queues, and so on)—nice way to run a systems software development group that still uses a one-way line editor.

All I want is the chance to use what I learned in school and to be able to do a programming project correctly. It would be such a treat.

Name withheld by request

Dear *DDJ*,

I am writing in response to Nick Turner's August Running Light about sloppy programmers. I am a programmer/engineer, and I see a lot of sloppy programming. In fact, I do some sloppy programming myself. I think I might have a clue as to what is going on.

Turner mentions the programmers who can "write entire operating system kernels . . . in one pass, in assembly code, that run perfectly the first time . . . ." Well, I think they are the exception to the rule. I do not, however, doubt the premise that we can all do it, it just takes the rest of us a little more time and concentration.

So why don't we take the time, concentrate, and code better? I propose that the reason why we don't is the reward system that most of us work within. My manager seems to be less concerned that a project works the first time than he does that it is done. He wants to "see something." Granted, I work for a defense contractor and my manager wants to record milestones, not finish projects. But I think this attitude is more prevalent than most of us think.

Why, then, does this attitude and reward system not affect the hot performers? I think it is because they, at some level, do not respond to the same reward system as the majority. Every organization has at least one programmer who walks to the beat of a different drummer. This is not to say that all individuals who fit this description are great programmers, but most of the great programmers I know (or know of) are of this type. On the other hand, this tends to make them more difficult to manage, partially because they do not respond to the reward system that reflects the views of management.

So what can be done? Well, we probably won't change the managers or their ideas of how things should be done. From my own experience, I find that if they want to see something I have two choices. First, I can slop it together and debug it later (when I have less time). Second, I can "stub" it off, perhaps only coding the user interface or some visual part of the program. This lets the manager see something, but the code he sees is good code. Later, I go back and, instead of debugging, I write (for the first time) the code that I stubbed off in the first place. This seems like the way to go, but it is sometimes difficult to determine when to stop coding and start stubbing.

So I haven't really offered a solution. Just some ideas as to what I think the probable cause is and what I conceive I should be doing to change the

Building an operating system is child's play . . .

. . . with Wendin's Operating System Toolbox™

We know.

We used it to build PCVMS, our $99 version of the versatile VAX/VMS operating system for the IBM PC.

And PCUNIX, the only operating system that puts the powerful features of UNIX on the PC for under $100.

But we didn't create this powerful software construction set just for us. The Toolbox is for any programmer who wants to build his or her own multitasking, multiuser operating systems.

Systems compatible with the PC-DOS file system and with most MS-DOS programs.

In nine basic modules, Toolbox provides everything you need (including source and object code) to build a personal operating system that fits you and your work habits — instead of the other way around.

And to help you get started, we've written a step-by-step instruction manual that shows you how to write a shell and link it with the Toolbox kernel.

The only thing we don't provide is a creative imagination. If you've got that, the rest is child's play.

Operating System Toolbox From Wendin. Only $99.

Ask about our other products for the IBM PC and true compatibles.

PCNX™

True multitasking, multiuser operating system similar to AT&T's popular UNIX™ operating system.

PCVMS™

Multitasking, multiuser version of DEC's powerful VAX/VMS operating system. Runs most MS-DOS programs.

XTC®

The ultimate programmer's editor. Multitasking macro language plus multiple linkable windows and buffers.

All products priced at $99 with source code included.

DEALER INQUIRIES WELCOME

Foreign orders inquire about shipping. Domestic orders add $5.00/1st item, $1.00 each additional item for shipping, handling, and insurance. We accept Visa/MC, American Express, COD, and Bank Drafts drawn on U.S. Banks.
Washington residents add 7.8% sales tax.

MS is a trademark of Microsoft, PC-DOS is a trademark of IBM. UNIX is a trademark of AT&T. VAX/VMS is a registered trademark of Digital Equipment Corporation.

Wendin and XTC are registered trademarks of Wendin, Inc. PCNX, PCVMS, Operating System Toolbox, and Personal Operating System are trademarks of Wendin, Inc.

ORDER HOTLINE
(509) 624-8088
(MON.-FRI., 8-5 PACIFIC TIME)

VISA MasterCard

WENDIN®
327 E. PACIFIC
SPOKANE, WA 99202

The people who make quality software tools affordable.

Circle **no. 112** on reader service card.

(continued from page 10)

situation. I will be watching with interest to see what other readers have to say about this problem.
Name withheld by request

Avoid Woe when Upgrading MS-DOS

Dear *DDJ*,
Allen Holub's "A Tale of Woe" (C Chest, September 1986) sparked some vivid memories of upgrading from MS-DOS 2.11 to MS-DOS 3.10. MS-DOS 2.11 and earlier would search for an available file from the beginning of the FAT each time. MS-DOS 3.10 is more efficient because it allocates from wherever it left off. This caused me much grief in trying to change the attributes and delete and replace the two system files Allen mentioned, IBMBIO.COM and IBMDOS.COM for the PC and IO.SYS and MSDOS.SYS for generic MS-DOS. These two files must reside in the first clusters of the disk. The first data cluster is cluster 2. Additionally, they must be contiguous. You can delete and replace these two files by following a few simple rules. The total number of clusters must be no more than the original unless there are some unallocated clusters just beyond these two files. Under MS-DOS 3.10, you can unhide and delete these files and then reboot from a floppy. This will restore the FAT pointer to start searching at the beginning of the disk to modify. You may then copy the system files, which will now be the first n clusters, and they will be contiguous. Now the disk will be bootable. It helps to be able to track through the FAT and directories if there is a problem or if more space is needed. So there may be no need to reformat.

Max G. Heffler
Landmark Graphics Corp.
1011 Hwy. 6 S, #120
Houston, TX 77077

OS-9 Continued

Dear *DDJ*,
I was dismayed to see the thrashing Heitzso gave to the OS-9 operating system in his October letter. I disagree with both his specific examples and his general conclusions (please pardon the assumption of gender).

To start with, OS-9 cannot compete with Unix on disk speed; Unix keeps part of its file structure in memory whereas OS-9 always keeps its sector allocation bit map on the disk up to date. There is a clear speed advantage to accessing information in memory rather than on disk, but you pay a price in vulnerability. In this case, Microware chose to make the file structure robust and corruptionproof.

Heitzso illustrates the "real problems" he has had with OS-9 by describing his difficulty in using the *tsleep( )* function. After reading his letter I wrote a simple C program to test the *tsleep( )* function, and I was unable to make it malfunction for any number of ticks I specified, over a range of 1 to several thousand. In every case the timing was +/− one tick, just as specified.

The *tsleep( )* function accepts a single parameter indicating the number of ticks to sleep. Most OS-9 systems use a tick granularity of 1/100 second, which also happened to be the length of time Heitzso wished the task to sleep. I suspect that he requested that the task sleep for one tick; however, the documentation clearly states that a tick parameter of 1 causes the calling task to give up its present time slice. At the expiration of the time slice, the task will be put back on the active process queue, where it will compete for CPU time with other processes that are ready to run. If the calling task gives up its time slice near the end of a quantum and there are no other executable processes, it is quite possible that *tsleep( )* will return after an interval as short as 1/3,000 second.

Heitzso also describes how the OS-9 Format utility has a bug that prevents the user from specifying a cluster size greater than one sector. In reality, the documentation for the Format utility mentions that at present only a cluster size of one is supported!

I have yet to uncover a bug in the operating system. Microware's latest product discrepancy report lists a single bug in the operating system components, and it is triggered by an obscure condition in a little-used system call.

I disagree with Heitzso's conclusion that Microware's customer support is poor. I have found Microware to be quite reasonable in the dealings I have had with it. Microware provides bug lists and work-arounds to those who request them, and it offers a special telephone hot line for professional software developers. One of my gripes is that the hot line is too expensive, but I have heard that this may change. I hope so.

During a time when many experts in the computing field were touting the use of high-level languages as the best way to create an operating system, Microware quietly crafted an elegant, modular, and extensible gem in assembly language. I look forward to seeing OS-9 dominate the 68000 market as more people recognize its merits.

Kurt Liebezeit
Ordinate Systems
505 W. Springfield
Champaign, IL 61802

We didn't do our homework when we published Heitzso's letter. This issue contains a look at the OS-9 operating system by Brian Capouch—eds.

Correction

Dear *DDJ*,
The Microsoft-supplied correction to Version 4 of the Microsoft Macro Assembler that Ray Duncan gives on page 96 of the September 1986 issue of *DDJ* is incorrect. The "correction" as published causes MASM to go off into never-never land. The error in the listing on page 96 is a typographical one. The byte entered into address *xxxx:72D4* should be *E9* instead of *39*. This error appears in the string of 34 bytes that are entered starting at *xxxx:72B8*.

Robert C. F. Bartels
P.O. Box 2240
Ann Arbor, MI 48106

DDJ

PC MAGAZINE EDITOR'S CHOICE

*. . . one package stands out as the best support available for Turbo Pascal programmers: **Blaise Computing's Turbo Power Tools.** This definitive set of prewritten Pascal functions and procedures will make the life of any programmer—from the beginner to the hard-core professional—easier and more productive.*

ANOTHER PLUS FROM BLAISE COMPUTING

Power Tools Plus™

Window Routines.
Memory Resident Routines.
Routinely.

The best just got better! Turbo POWER TOOLS, acclaimed as the best programmer support package for Turbo Pascal, now has even more functions, more detailed documentation and more sample programs.

NO SECRETS

Turbo POWER TOOLS PLUS is crafted so that the source is efficient, readable and easy to modify. We don't keep secrets! We tell you exactly how windows are managed, how interrupt service routines can be written in Turbo Pascal, and how to write memory resident programs that can even access the disk. Maybe you've heard of some undocumented DOS features that resident programs use to weave their magic. Turbo POWER TOOLS PLUS documents these features and lets you make your own magic!

Here's just part of the PLUS in Turbo POWER TOOLS PLUS:

- ◆ **WINDOWS** that are stackable, removable, with optional borders and a cursor memory.
- ◆ **FAST DIRECT VIDEO ACCESS** for efficiency.
- ◆ **SCREEN HANDLING** including multiple monitor and EGA 43-line support.
- ◆ **POP-UP MENUS** which are flexible, efficient and easy to use, giving your applications that polished look.
- ◆ **INTERRUPT SERVICE ROUTINES** that can be written in Turbo Pascal without the need for assembly language or inline code.
- ◆ **INTERVENTION CODE** lets you develop memory resident applications that can take full advantage of DOS capabilities. With simple procedure calls, you can "schedule" a Turbo Pascal procedure to execute either when a "hot key" is pressed, or at a specified time.
- ◆ **PROGRAM CONTROL ROUTINES** allow you to run other programs from Turbo Pascal, and even execute DOS commands.
- ◆ **MEMORY MANAGEMENT** allows you to monitor, allocate and free DOS-controlled memory.
- ◆ **DIRECTORY AND FILE HANDLING** support to let you take advantage of the newer features of DOS including networking.
- ◆ **STRING** procedures allowing powerful translation and conversion capabilities.
- ◆ **FULL SOURCE CODE** for all included routines, sample programs and utilities.
- ◆ **DOCUMENTATION, TECHNICAL SUPPORT** and attention to detail that have distinguished Blaise Computing over the years.

Turbo POWER TOOLS PLUS** supports Turbo Pascal Version 2.0 and later and is just **$99.95.

*Another quality product from Blaise Computing: **Turbo ASYNCH PLUS™***

A new package which provides the crucial core of hardware interrupt support needed to build applications that communicate. ASYNCH PLUS offers simultaneous buffered input and output to both COM ports at speeds up to 9600 baud. The XON/XOFF protocol is supported. Now it also includes the "XMODEM" file-transfer protocol and support for Hayes compatible modems.

The underlying functions of Turbo ASYNCH PLUS are carefully crafted in assembler for efficiency and drive the UART and programmable interrupt controller chips directly. These functions, installed as a runtime resident system, require just 3.2K bytes. The high level function are all written in Turbo Pascal in the same style and format as Turbo POWER TOOLS PLUS. All source code is included for just $99.95.

BLAISE COMPUTING INC.

2560 Ninth Street, Suite 316 Berkeley, CA 94710 (415) 540-5441

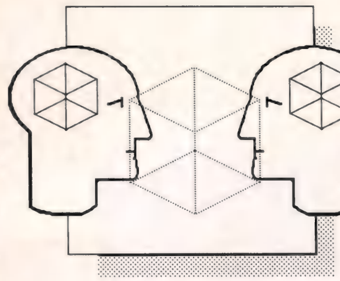
ORDER TOLL-FREE 800-227-8087

Calif. residents call (415) 540-5441

YES, send me the PLUS I need! Enclosed is $______ for
☐ Turbo POWER TOOLS PLUS ☐ Turbo ASYNCH PLUS ☐
OTHER ______. (CA residents add 6½% Sales Tax. All domestic orders add $10.00 for Federal Express shipping.)
Name: ______ Phone: (___) ______
Shipping Address: ______
City: ______ State: ____ Zip: ____
VISA or MC #: ______ Exp. Date: ____

Circle no. **159** on reader service card.

VIEWPOINT



Logic and PROLOG

The touted virtue of PROLOG is that it provides a basis for programming in logic—hence its name. This suggests that logically correct descriptions or axiomatizations of a body of knowledge can be transcribed into PROLOG and that the appropriate deductions could then be drawn by PROLOG's inference engine. The idea of systematization of knowledge by way of postulates embedded in a deductive system has proven to be a powerful one since the time of Euclid. With the development of predicate logic by Frege and Russell, the idea received new impetus in this century. Many areas of mathematical and scientific investigation have axiomatic foundations—set theory being a prime example. The expansion and codification of knowledge by the deductive-axiomatic method, as the central methodology of knowledge, has its critics. But what has delayed its use outside theory construction itself is that, for practical real-time applications, hand-deduction from a knowledge base is too

by Dick Butrick

slow. What PROLOG promises is speed of deduction. You could simply query a knowledge base (axioms, postulates), and PROLOG would make deductions with computational speed—giving tremendous impetus to the deductive-axiomatic method as the central methodology of knowledge by removing the practical barrier to its use.

Unfortunately, the match between formal logic and PROLOG is tenuous at best. In fact, from a strictly logical point of view, PROLOG is inconsistent. Because any inconsistent deductive system is complete (if you can derive a contradiction, you can derive anything), PROLOG is theoretically complete. In implementation, however, PROLOG is not just inconsistent, it is incomplete. Time considerations and stack space are not considerations in pure logic, but these realities render PROLOG radically incomplete.

Of course, there are PROLOGs and PROLOGs. Specific reference here is to PLS PROLOG. However, the points raised apply just as well to Borland PROLOG and indeed to any nonpure, expanded, DEC-20-type PROLOG (Quintus PROLOG, CPROLOG, Poplog, and so on).

Typically, in testing out PROLOG as a deductive-axiomatic shell, a logician might enter postulates for the transitivity of *R*:

```
(x) (y) (z) (Rxy & Ryz --> Rxz)
```

In the Simple syntax of PLS, this becomes:

```
R(x z) if R(x y) and R(y z)
```

Given the *R*-facts *R(a b)* and *R(b c)*, it seems reasonable to query the system with *is(R(a c))*? PROLOG promptly responds with "no more space." Stack overflow has occurred.

This is not apt to impress the logician, or for that matter the layman, with the deductive power of PROLOG. At this point the dismayed logician might enter the PROLOG equivalent of *{p <--> q, p}* and query the system for *q*. The PROLOG equivalent is:

```
p if q
q if p
p
```

And the query is *is(q)*. "No more space" is again the reply.

At this point the logician might wonder if the inference engine is out of gas. In fact the system is in a goal-reduction loop. It reasons thus: To get *q*, first get *p*; to get *p*, first get *q*; to get *q*, first get *p*; and so forth. It never gets beyond the first two rules. In fact, *{p if p, p}* along with the query for *p* will put the system into an infinite loop.

This might seem a simple problem to solve, which of course it is for specific cases. In general, however, it can be shown to be unsolvable. A loop-detection procedure for a goal-reduction theorem prover is equivalent to the halting problem shown by Alan Turing in the 30s to be unsolvable. Essentially, a loop-detection monitor requires more logic than the goal reduction it monitors.

To make matters worse, the "no space left" response can be triggered without setting up a goal-reduction loop: *{H(x y) if H(x x), H(a a), H(b b)}* along with the query *is(H(a b))* causes stack overflow. In Borland PROLOG, *{H(x y if H(y x), H(a b)}* along with the query for *H(b a)* does the trick. If you cannot enter the postulates declaring the commutativity of a relation, then you might question whether PROLOG belongs in the remedial logic class for absolute dummies.

Examples such as the foregoing are legion, but they merely demonstrate the radical incompleteness of PROLOG. What is even more insidious is PROLOG's inconsistency. The treacherous dimension to PROLOG is not so much what it can't do as what it can. Consider the following deduction, based on the deduction rule known to logicians as mirabile dictu:

```
p if not q
q                          / Hence not p
```

Querying this little postulate set with *is(not p)* yields the astounding answer "yes." Not only that, the system will make further deductions on the basis of this fallacious deduction. Adding the postulate *r if not p* and querying the system for *r* yields the answer "yes." Again, such examples are legion.

Borland refers to PROLOG's treatment of negation as "novel." The justification for this novel treatment of negation is, according to Borland, the hidden assumption of the law of the excluded middle automatically made by PROLOG. Thus for the one-place predicate *h*, PROLOG automatically assumes *h(x)* or *not h(x)*. The latter is

(continued on page 138)

Dick Butrick, Ohio University, Athens, OH 45701. Dick is a professor in the Computer Science Department.

680xx Computers: Where Are They Going?

by Nick Turner

The 68000 led to the first powerful sixer machine.

Since the beginning of the microprocessor industry, two groups of programmers have emerged—a result of the divergence between those who liked the 8080 and those who liked the 6502. The 8080 had dedicated I/O instructions that used a separate address space, and it had several fairly specialized internal registers. Its instruction set was designed to be powerful and specific. To learn it programmers had to memorize its specialized instructions and address modes. Conversely, the 6502's instructions were more general purpose but less powerful. The instructions were easier to learn, but it took more of them to accomplish the same tasks. Its I/O was memory-mapped, making input and output identical (from the processor's point of view) to normal memory addressing. Memory-mapped I/O also reduced the number of specialized instructions that had to be learned. The two camps began to diverge as early as 1974, and by the early 80s, those programmers who were most fervent acquired the nicknames "eighters" and "sixers."

When the Z80 was introduced, the eighters suddenly had a much more powerful tool. The sixers had the 6800, and then the 6809, but there weren't many viable machines that used those chips. Ardent sixers had to be content with their Apple IIs and Commodore Pets and 64s. Certainly, some really magnificent work came out on the Apple II during the early 80s, showing that the Volkswagen of personal computers was a lot more powerful than had been thought. Toward the end of this period, some manufacturers tried to get the best of both worlds—for example, the OSI Challenger III, a strange machine with a Z80, a 6502, and a 6800 all in the same box.

For several years, it seemed likely that the Intel line of CPU chips would eventually take over the market. Many devoted sixers shuddered when the IBM PC turned out to have an eighter chip as its heart.

It wasn't until the 68000 appeared that there was a really powerful sixer machine. What led to the development of the 68000? Motorola's engineers perceived the need for a much more general instruction set so that the chip design could be cleaner and easier to mask. They wanted a CPU that would be upward compatible with future, more powerful chips, without the need for expensive "modes" that hamper functionality and take up space on the chip. For the most part, they seem to have succeeded. Though the 68000 does have some disadvantages and departures from a truly general-purpose design (such as the inability to store data using PC-relative addressing), it's a far cry from the restrictive modes and specializations of the high-end eighter chips. Motorola took an enormous gamble in introducing what was projected to be a whole series of CPUs with a completely new instruction set. Low-level programs had to be rewritten for the 68000. There was (and still is) a lot of momentum in the Intel line of chips. Has Motorola's gamble paid off?

Today's 680xx Line

To me the most valuable feature of the 68000 line, aside from the philosophy behind it, is the enormous range of speed and power available. With few or no changes to your software, you can move up from the 68008, which is roughly comparable to a fast Z80 in power, all the way to

Nick Turner, 501 Galveston Dr., Redwood City, CA 94063. Nick is a DDJ editor.

a 20-MHz 68020, which easily outperforms a small VAX.

Further, the line includes a wide and constantly growing selection of support chips. Most of the support chips are currently made by Motorola, but a growing number are being designed by other companies expressly for the 68000 line (an indication of the health of the 68000 standard). The most important support chips are the 68851 MMU and the 68881 FPU, both of which are true coprocessors when used with the 68020 (or later) CPU.

Perhaps as a result of the ease of designing with the 68000, a new flock of 68000-based computers has appeared in the last three years. Table 1, page 18, shows a sampling of the range of power and speed in the line. Of course, there are the relatively low-price personal systems, most of which have graphics-oriented interfaces. But you also have high-end workstations, such as the Sun systems and the Apollo Domain network, and a host of other high-end systems, including some powerful image processing equipment. The VME bus, originally designed around the 68000 line, has rapidly become a worldwide standard for rugged, powerful, modular computer systems. It's supported by an international coalition of companies, the VME International Trade Association (VITA), and is one of the most carefully defined system specifications I've encountered.

Some of the most interesting systems are the most recent. For example, the Mustang-020, a 68020-based system, has been available since last spring. It's amazingly powerful for its price, and it shows great promise as a general-purpose industrial machine. One of its chief assets is its size—in a box no larger than an IBM PC, it provides more power than do many minicomputers. Another system to watch is the Quantum QL, which comes from Sinclair in England. According to some preliminary literature, the entire computer is contained within a box the size of an IBM PC keyboard, and it runs a multitasking operating system called QDOS.

Software Issues

Because of its generalized and relatively simple design, the 68000 works well with multitasking operating systems—for example, Unix runs well under the 68020. Another interesting operating system that has recently been increasing in popularity is OS-9. Originally designed for the 6809, OS-9 is fast, small, and most important, highly accessible. (See Brian Capouch's article on OS-9 on page 30.)

The 68000 line also lends itself well to graphics-oriented interfaces because of its efficient handling of large memory spaces and its ability to easily accommodate large memory-mapped I/O spaces. The Macintosh operating system, for example, has set a new user interface standard for personal computers. Although the Mac OS (especially in its original incarnation) contained some major flaws, particularly in the area of disk organization, it has been much imitated, and most of the flaws have been corrected with the release of Apple's Hierarchical File System. The Amiga and Atari ST both have similar "desktop" screens, icons, and mice.

In the area of languages, the most prominent has been C. Table 2, page 18, illustrates some typical execution times of programs compiled in C on 68000-based machines. The benchmarks in the table were chosen to illustrate a variety of operations roughly representative of actual programs. C is in some ways perfectly matched to the 68000 line—the regularity and generality of the instruction set makes a language such as C almost a natural. Because of its relatively low-level nature, C translates readily into well-defined groups of machine instructions.

The Future

The 68000 line is the most serious competition for the Intel line, and the machines that use 68000s are varied and colorful. But will the 68000 line continue to grow and diversify? Will the industry support new members by

(continued from page 17)

creating products that incorporate them? I think so, for several reasons.

First, Motorola seems to have latched firmly to the idea that object-code compatibility is a must in the 68000 line. So far, each of the new CPUs has been almost completely compatible with its predecessors. The only exceptions come when you're writing time-slicing OS code or you have to deal with interrupts. Typically, that sort of code is fairly easy to upgrade, and the actual applications (if they're written intelligently) seldom require any changes.

Second, you can expect continued growth and diversity in the 68000 line. Some preliminary information on the 68030 processor has just landed on my desk. It will have not only the instruction cache of the 68020 but also a data cache, a memory manager, and much more. The result will be a CPU that has the 68020's speed in tight loops (because of the instruction cache) and that also can at times execute completely on-chip for extended periods (because frequently accessed data will be in the cache). The rumored 68040 may use a 64- or 128-bit data bus for reads (although it will still use a 32-bit bus for writes). It may contain an even larger RAM cache as well, plus a more powerful memory manager. In the peripherals department, the 68882 will be a more powerful, pin-compatible version of the 68881 floating-point coprocessor that, because of internal multiprocessing, will significantly outstrip the performance of its predecessor.

Finally, there are rumors of many other support chips floating around, along with some really fun rumors about 68100 or 68200 CPU chips. For example, how about a chip containing four 68030 equivalents, all executing simultaneously from dual, dynamically allocated 16K instruction and data caches? Of course, that's nothing more than a nasty rumor. There's probably no truth to it whatsoever.

A Dream Machine?

Will future developments live up to the expectations of software developers? So far the 68000 line has done well in a market that might otherwise have been completely dominated by Intel. The 80386 is strong competition, but the 68030 sounds like a programmer's dream. Unless some company takes over the sixer marketplace within the next few years, it looks like the 68000 and the more powerful related chips will prevail as the premier sixer CPUs.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

Type	Maker	Model	Base Prices	Memory Range (Mbytes)	CPU	Clock Speed (MHz)	OS	Open Design	Intro Date
Portable	Sinclair	QL	$266	.5	68008	8?	QDOS	No	1986
Macalikes	Atari	ST 520	$800	.5-1	68000	8	TOS	No	7/85
	Commodore	AMIGA 1000	$1,000–$3,000	.25–4	68000	7.16	AmigaDOS	Yes	10/85
	Apple	Macintosh Enhanced	$1,700	.25-4	68000	7.8	Mac OS	No	4/86
		Macintosh Plus	$2,200	1–4	68000	7.8	Mac OS	No	1/86
		"Jonathon"	?	2–?	68020?	16?	Mac OS? Unix?	Yes	3/87?
Workhorses	Data-Comp	Mustang-020	$4,000	2	68020	12–16	OS-9	No	3/86
	Various	VME bus	$4,000–$20,000	.5-16	(various)	8–20	(various)	Yes	—
Workstations	Apollo	Domain	$9,900–$70,000	2–16	68020	12–20	Unix	Yes	2/86
	Sun	Sun II	$19,900	1–4	68010	10	Unix 4.2	Yes	11/83
		Sun III	$19,900	4–16	68020	16.7	Unix 4.2	Yes	9/85

Table 1: *Comparison of representative 680xx systems*

Benchmarks:

System	Compiler	Looptst (500)	Pointer (1500)	Fibtest (18)	Sieve (140)
Amiga	Lattice	44.3	37.5	48.8	81.7
	Manx Aztec 16	29.4	33.3	35.1	64.3
	Manx Axtec 32	38.4	35.1	40.1	80.7
Atari ST	Aleyon	25.6	28.4	31.2	56.5
	Lattice	30.7	30.7	35.0	62.3
	Mark Williams	30.7	30.0	37.2	60.1
	Megamax	25.6	35.3	33.3	53.9
Macintosh	Lightspeed	37.4	42.3	45.4	78.9
IBM PC	Microsoft C	25.8	30.9	37.1	60.1

Table 2: *Representative C compiler benchmarks (in seconds)*

Unleash The Most Powerful Development Tools On The Planet DOS.

UNIFY DBMS/DOS. The UNIX World Leader Brings A New Dimension To DOS Application Development.

What happens as the DOS world expands? As a new generation of hardware takes over? As networking becomes more important? The potential is enormous. But until now, the tools to achieve it have been limited.

Now a leader from another world unleashes that potential: UNIFY® DBMS. The leading relational DBMS in the UNIX™ world. And now, the most advanced set of application development tools in the DOS world.

With UNIFY DBMS, DOS developers have new power to build more sophisticated applications than ever before possible.

The power to write high performance "C" programs that will access the data base, using Unify's Direct Host Language Interface.

The power of an industry standard query language—SQL.

The power of unmatched speed in production applications. Only UNIFY DBMS is specifically engineered for transaction throughput. With unique performance features like PathFinder™ Architecture multiple access methods, for the fastest possible data base access.

The power of comprehensive program development and screen management tools. Plus a state-of-the-art fourth generation report-writer.

What's more, with UNIFY DBMS, the potential of networked applications becomes a reality. Unlike DBMS systems which were originally single-user (and which have a long stretch to accommodate more users), UNIFY DBMS is a *proven* multi-user system.

And because UNIFY DBMS/DOS is the best of two worlds, it offers you the most powerful benefit of all: DBMS applications that can grow as your needs grow. From single user DOS. To networked DOS. To multi-user UNIX. All without changing your applications.

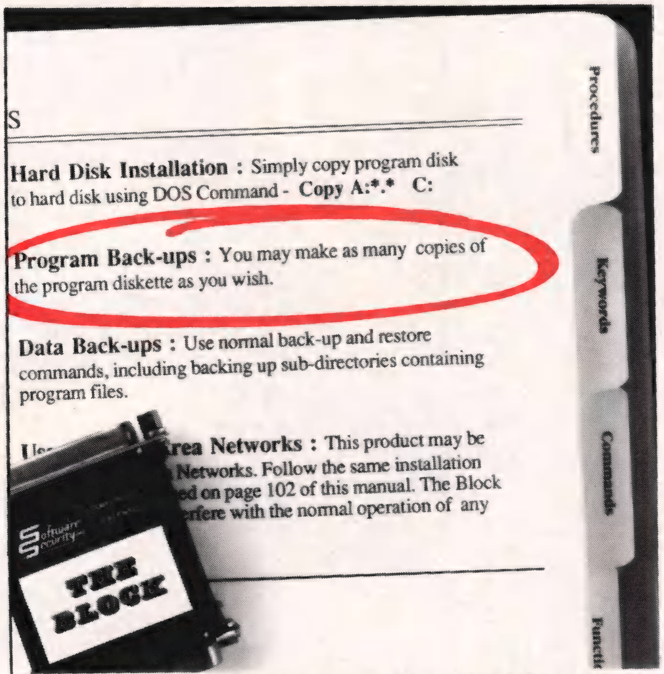
Call the Unify Information Hotline for our free booklet: The New DOS World. (503) 635-7777

UNIFY
CORPORATION
4000 Kruse Way Place
Lake Oswego, OR 97034

Circle **no. 332** on reader service card.

"How to protect your software by letting people copy it."

By Dick Erett, President of Software Security



Soon all software installation procedures will be as straightforward as this. The only difference will be whether you include the option to steal your product or not.

Inventor and entrepreneur, Dick Erett, explains his company's view on the protection of intellectual property.

"A crucial point that even sophisticated software development companies and the trade press seem to be missing or ignoring is this:

Software protection must be understood to be a distinctively different concept from that commonly referred to as copy protection.

Fundamentally, software protection involves devising a method that prevents unauthorized use of a program, without restricting a legitimate user from making any number of additional copies or preventing program operation via hard disk or LANs.

Logic dictates that magnetic media can no more protect itself from misuse than a padlock can lock itself.

Software protection must reside outside the actual storage media. The technique can then be made as tamper proof as deemed necessary. If one is clever enough, patent law can be brought to bear on the method.

Software protection is at a crossroads and the choices are clear. You can give product away to a segment of the market, or take a stand against the theft of your intellectual property.

"...giving your software away is fine..."

We strongly believe that giving your software away is fine, if you make the decision to do so. However, if the public's sense of ethics is determining company policy, then you are no longer in control.

We have patented a device that protects your software while allowing unlimited archival copies and uninhibited use of hard disks and LANs. The name of this product is The BLOCK™.

The BLOCK is the only patented method we know of to protect your investment. It answers all the complaints of reasonable people concerning software protection.

In reality, the only people who could object are those who would like the option of stealing your company's product.

"...eliminating the rationale for copy-busting..."

Since The BLOCK allows a user to make unlimited archival copies the rationale for copy-busting programs is eliminated.

The BLOCK is fully protected by federal patent law rather than the less effective copyright statutes. The law clearly prohibits the production of work-alike devices to replace The BLOCK.

The BLOCK attaches to any communications port of virtually any microcomputer. It comes with a unique customer product number programmed into the circuit.

The BLOCK is transparent to any device attached to the port. Once it is in place users are essentially unaware of its presence. The BLOCK may be daisy-chained to provide security for more than one software package.

Each software developer devises their own procedure for accessing The BLOCK to confirm a legitimate user. If it is not present, then the program can take appropriate action.

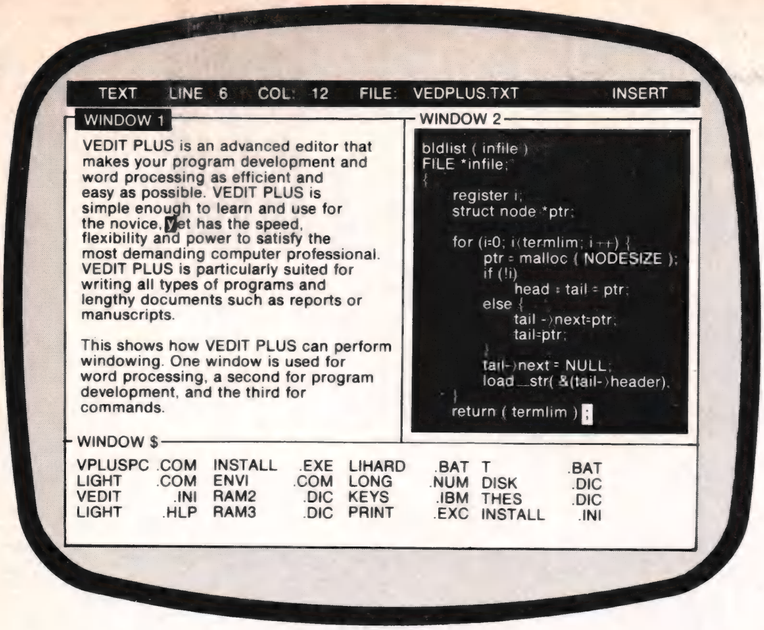
"...possibilities... limited only by your imagination..."

The elegance of The BLOCK lies in its simplicity. Once you understand the principle of The BLOCK, hundreds of possibilities will manifest themselves, limited only by your imagination.

Your efforts, investments and intellectual property belong to you, and you have an obligation to protect them. Let us help you safeguard what's rightfully yours. Call today for our brochure, or a demo unit."

Software Security inc.

870 High Ridge Road Stamford, Connecticut 06905
203 329 8870



VEDIT PLUS

#1 CHOICE IN PROGRAMMABLE EDITORS

VEDIT PLUS has been the #1 choice of professionals since 1980. Our latest release is even better - you can open windows to simultaneously edit several files, access many editing functions with pop-up menus, use keystroke macros to speed editing, and run other programs or DOS commands from within VEDIT PLUS.

Whether your needs are program development, technical writing or word processing, VEDIT PLUS is your answer. With over 40,000 users you can depend on VEDIT PLUS to perform consistently and reliably. It is simple enough to learn for the novice, yet has the speed, flexibility and power to satisfy the most demanding professional.

Compare. No other editor is as powerful - unlimited keystroke macros, instant 'off-the-cuff' command macros utilizing a complete programming language, single command file comparison, special word processing and programming features. No other editor is as easy to use - on-line help, pop-up menus, 75 page tutorial, 380 page manual, and VEDIT PLUS is completely customizable.

Fully supports color windows on IBM CGA & EGA, and even windows on most CRT terminals (including CRT's connected to an IBM PC). Available for IBM PC, TI Professional, Tandy 2000, DEC Rainbow, Wang PC, MS-DOS, CP/M-86 and CP/M-80. Order direct or from your dealer. $185

> "To sum things up, VEDIT PLUS is a small, fast, sophisticated editor with a wealth of features and a good macro language. It offers many rewards for the dedicated programmer."
> **Computer Language, Chris Wolf, Scott Lewis, Mark Gayman 6/86**

> "VEDIT PLUS is a wholly remarkable program: blindingly fast, extremely powerful, and highly flexible."
> **Profiles Magazine, Robert Lavenda 4/86**

VEDIT PLUS FEATURES

- Simultaneously edit up to 37 files of unlimited size.
- Split the screen into variable sized windows.
- 'Virtual' disk buffering simplifies editing of large files.
- Memory management supports up to 640K.
- Execute DOS commands or other programs.
- MS-DOS pathname and CP/M user number support.
- Horizontal scrolling - edit long lines.
- Flexible 'cut and paste' with 36 text registers.
- Customization - determine your own keyboard layout, create your own editing functions, support any screen size, any CRT.
- Optimized for IBM PC/XT/AT. Also 132 column & up to 70 lines.

EASY TO USE

- Interactive on-line help is user changeable and expandable.
- On-line integer calculator (also algebraic expressions).
- Single key search and global or selective replace.
- Pop-up menus for easy access to many editing functions.
- Keystroke macros speed editing, 'hot keys' for menu functions.

FOR PROGRAMMERS

- Automatic Indent/Undent for 'C', PL/I or PASCAL.
- Match/check nested parentheses, i.e. '{' and '}' for 'C'.
- Automatic conversion to upper case for assembly language labels, opcodes, operands with comments unchanged.
- Optional 8080 to 8086 source code translator.

FOR WRITERS

- Word Wrap and paragraph formatting at adjustable margins.
- Right margin justification.
- Support foreign, graphic and special characters.
- Convert WordStar and mainframe files.
- Print any portion of file; separate printer margins.

MACRO PROGRAMMING LANGUAGE

- 'If-then-else', looping, testing, branching, user prompts keyboard input, 17 bit algebraic expressions, variables.
- CRT emulation within windows, Forms entry.
- Simplifies complex text processing, formatting, conversions and translations.
- Complete TECO capability.
- Free macros: • Full screen file compare/merge • Sort mailing lists • Print Formatter • Main menu

Circle **no. 122** on reader service card.

CompuView

1955 Pauline Blvd., Ann Arbor, MI 48103 (313) 996-1299, TELEX 701821

A Mini Forth for the 68000

by G. Yates Fletcher

The inside approach to Forth makes the language much more palatable.

My exposure to Forth has been limited to the proselytizing of a few Forth fanatics, articles in various publications, and a fairly careful reading of Leo Brodie's excellent book *Starting Forth* (Englewood Cliffs, N.J.: Prentice-Hall, 1981). One of the attractions of the language for me is its fundamental simplicity. The interpreter, threader, and kernel of basic supporting words are small enough that (at least in theory) a single programmer can create a working system with a moderate amount of effort. I often toyed with the notion of doing it myself just for the learning experience (read that fun), but I never got to the point of making a serious attempt until last fall when I was assigned to teach for the umpteenth time our sophomore-level course in computer organization and assembly language. I was casting about for some way to generate enthusiasm, which although not a prerequisite for teaching seems to make the process more enjoyable for all concerned. It struck me that building a Forth system might make good programming exercises for my students. Thus was FLINT born.

I thought, what the heck, I'll write a "no frills" Forth and give the students the executable code (so they can play with it and see how it's supposed to work) and the source code minus the modules they will be required to write. By the time I've taught them enough about assembly-language programming to do the job and enough about Forth that they understand what is required, the semester should be winding down. Amazingly enough, everything went pretty much according to plan (albeit with considerably more sweat than I anticipated). As you might have guessed by now, the instructor probably learned a lot more than the students, but that's one of the reasons why I took the job.

G. Yates Fletcher, Dept. of Computer Science, North Carolina State University, P.O. Box 8206, Raleigh, NC 27695-8206. Mr. Fletcher is an assistant professor of computer science.

The product of this labor is not a standard Forth. As the venture was educational/recreational and not commercial, I never bothered to find out what the standards were or even look at the code for a real Forth. Thus I have chosen the acronym FLINT, for Forth-like interpreter and threader, to describe the system. FLINT was basically reverse-engineered from Brodie's description of the language, so the implementation is probably a blend of novelty and naiveté. Nevertheless, I feel that it is more than a toy, as I have used it to write a turtle-graphics program for my terminal; a full-screen editor (for Forth screens, of course); and a version of a standard prime-number seive benchmark (*Byte*, January 1983) that runs in a little more than 20 seconds on an 8-MHz 68000, making it faster than any of the microcomputer Forths listed.

I can conceive several levels at which this program may be useful. For those not familiar with Forth, it might help to be an introduction. It is no substitute for a good book such as Brodie's, but it could make a worthwhile companion. Those approaching Forth from the outside as another language are often put off or puzzled by its many idiosyncracies. The inside approach to Forth as a program gives a complementary perspective that resolves many of these mysteries and makes the language much more palatable. For do-it-yourselfers this program is evidence that novices can indeed produce something workable. They can read enough of the description and documentation to get a good idea of what needs to be done and blast off on their own, referring to the code perhaps when stuck or merely to confirm that their own way of doing things is better. Forth programmers who don't have a version running on their own 68000 machine might be able to revise and polish it up enough so that it resembles some usable standard. Forth cognoscenti might be interested to know that FLINT produces what I have since learned is subroutine threaded code (which seems to be particularly appropriate for the 68000) as opposed to the more usual indirect threaded code.

Overall Structure

FLINT was written with several goals in mind: to provide students with an example of a well-structured, real world, assembly-language program; to illustrate the utility and power of the 68000 instruction set and addressing modes; and to test the theory that Forth is more naturally understood as a program rather than as a language.

I've made a fairly serious attempt to structure and comment the code properly. Because everything is in

there somewhere, much can be learned (at least in theory) by reading the program carefully. In fact, several of my students have intimated (very discreetly) that they found the code much less confusing than my explanations of it. The 68000 assembly code comprises a minimal kernel of less than 500 lines and is arranged as shown in Table 1, below.

FLINT requires some basic BIOS support from the host in the form of macros that users must tailor for their own machines and resident system software. My system is a Sage II (now Stride), which has two floppy drives, 512K RAM, and 64K PROM and runs on an 8-MHz 68000. The PROM contains all the necessary BIOS support as well as a monitor/debugger that furnished an excellent environment for developing and running FLINT. Macro definitions for my system are in Listing One, page 52.

Interpreter and Dictionary

The token is the basic unit processed by the interpreter. Tokens are not obtained directly from the terminal but are taken from an 80-character line buffer. After prompting the user, routine *LINE* (Listing Two, page 52) fills the line buffer from the terminal and terminates upon receiving a carriage return character. *TOKEN* takes its input from the buffer and recognizes the blank as a token delimiter. The carriage return embedded in the buffer (the one that terminated *LINE*) is seen as a legal token whose execution returns control to *LINE*. In support of its buffer-filling function, *LINE* recognizes and handles backspaces.

The interpreter is composed of several routines whose operation forms an instruction cycle fed by the user interactively. The instructions are small modules of code called words whose actions are normally directed at data (or pointers to the data) residing on a parameter stack. The words are identified by short alphanumeric strings (tokens) that are symbols or English words that usually describe or are related to the action they perform. The words are arranged in a linked list in which each node contains the identifying token, a link pointer, and the code defining the word's action. This linked list is called the dictionary.

A dictionary entry starts with the identifier, which is a 4-byte value. It consists of the length and the first three characters of the name of the word being identified. For example, if the word were *EXECUTE*, the identifier would contain a 7 and then the characters *EXE*.

The next field is the link pointer, which is a 2-byte field containing the address of the previous word's identifier. Thus a dictionary search always starts with the most recently defined word and works backward. For larger systems you might want to make the link pointer a 4-byte value or perhaps make it a relative value instead of an absolute address.

After the link pointer is the code field, which contains the machine code that runs when the word is executed.

The outer interpreter is a loop that waits for and accepts input tokens, searches the dictionary until a match is found, and extracts the address of the corresponding code. The address can be sent to the inner interpreter for direct execution. If a token is not in the dictionary, it is assumed to represent a number in the current base, and its value is extracted and placed on the stack. If this attempt fails, an error is assumed and the *WHAZZAT* token is invoked.

Words can be defined as well as executed. The interpreter has an alternate compile mode (the standard mode is execute mode). The interpreter's primary task remains that of extracting the code address of an input token. When in execute mode, a *JSR* to this address is executed as before. But when the interpreter is in compile mode, a *JSR* to the code address is written into the dictionary. Execution is deferred until the word being defined is invoked in execute mode. Any number encountered in compile mode is handled by generating code in the dictionary that will push the value onto the stack at execution time. These compiled numbers are called literals. Additional support is provided for a submode in which words can be defined directly in machine code.

How FLINT Works

For a concrete example of how FLINT works, let's look at the activity that occurs as the first word in the inner shell, *CONSTANT*, is defined. Envision using this word interactively as follows:

```
10 CONSTANT TEN
```

or

```
12 CONSTANT DOZEN
```

CONSTANT will thus be invoked to define words that are constants. If, for example, you invoke the word *TEN*, you will expect it to push the number *10* onto the stack; if you invoke *DOZEN*, you expect *12* to be pushed; and so on. Let's call *TEN* and *DOZEN* instances of *CONSTANT*. Thus when *CONSTANT* is invoked, it will take the value of the instance from the stack and the name of the instance from the input stream.

Now look at the definition of *CONSTANT* to see how this activity is to be directed. The defining string is

```
: CONSTANT TOKEN HEADER LITERAL
                    CODE 3AFC 4E75 ;
```

The outer interpreter, operating in execute mode, picks up the token ":", finds a match in the dictionary, and proceeds to execute it. Examining the code for ":", you see that a subroutine call (*JSR*) to *TOKEN* will be excuted. The action of *TOKEN*, of course, is to pull in the next token from the input stream, and in this case it will be the token *CONSTANT*. The next *JSR* is to *HEADER*, whose action is to produce a dictionary header for the newly cap-

Outer interpreter
Line-buffer input routines: *PROMPT LINE*
Words supporting execute mode: *TOKEN, SEARCH, NUMBER, EXECUTE, WHAZZAT*
Words supporting compile mode: *COMPILE, HEADER, IMMEDIATE, ":", CODE, ";", ",", LITERAL*
System variables: *BASE, CBLOCK, EDBUF, DICT*
Words supporting disk I/O: *LOAD, GO, SAVE, INTERACTIVE*
Words supporting terminal output: *TYPE, ".", ".S"*
Miscellaneous words: *"(", QUIT, LOGOFF*

Table 1: *Arrangement of 68000 assembly code in kernel*

tured token *CONSTANT*. The remaining activity initiated by the definition will produce its code field.

The last activity of ":" is to set the compile flag, *FCOLON*, and return to the outer interpreter. The interpreter pulls the next token, *TOKEN*, from the input stream and extracts its code address. Because the interpreter is now in compile mode, a *JSR* to *TOKEN* will be written in the dictionary—that is, into the code field of *CONSTANT*. This means that the first activity of *CONSTANT*, when it is itself executed (called execution-time behavior), will be (surprise, surprise) to pull in the name of the instance from the input stream. In like manner a *JSR* to *HEADER* will become the next part of the code for *CONSTANT*, so now the execution-time behavior of *CONSTANT* has been defined up to the point where a header for the instance has been created. It is now time to define the execution-time behavior of *CONSTANT* that will define the execution-time behavior of its instances. Thus you must direct *CONSTANT* to take the value of its instance from the stack and write code in the dictionary that upon execution will place this value on the stack. This is an exact description of what *LITERAL* does, so its inclusion in the definition solves the problem neatly.

The activity *CONSTANT* must perform that has not yet been coded is to close the definition of its instance. At this point in the definition process, the outer interpreter, in compile mode, picks up the token *CODE*. A careful examination of the header for *CODE* shows that its listed token length is 132, which is 128 more than it should be. This is no accident. It is in fact the manner in which words are tagged as immediate, meaning that they are to be executed even when the interpreter is in compile mode. The necessity for such words should be obvious because otherwise, for instance, there would be no good way to terminate a definition. (When *TOKEN* picks up an immediate word, it sets the immediate flag, *FIMMED*, to let the interpreter know that the word is to be executed.) *CODE* sets the system base to 16 (hex); sets the code flag, *FCODE*; and returns to the interpreter.

When the tokens *3AFC* and *4E75* are picked up, they are not found in the dictionary. Thus each one is sent in turn to *NUMBER*, which extracts its proper hex value and places it on the stack from where the interpreter (now in code mode) copies them into the dictionary. The code *3AFC 4E75* translates as *MOVE.W #04E75H,(A5)+*. Thus the final activity in the run-time behavior of *CONSTANT* is to copy *04E75H* into the dictionary definition of the instance. Because *4E75* is the code for *RTS*, this activity will effectively close the definition of the instance. All that remains is to close the definition of *CONSTANT*. This is the job of ";", which resets the system base to 10, clears the compile and code flags, and writes an *RTS* into the dictionary.

Well, there you have it: just a simple little definition. Readers new to Forth (if there are any who have made it this far) probably feel that the claims for its simplicity are highly exaggerated, and even those with some familiarity may find themselves a bit glazed over. The bad news is that the operation of the FLINT compiler is often a fairly complicated activity. The good news is that it seldom gets any more complicated than the example given here. Complexity is after all a relative thing. Would anyone care to write a step-by-step explanation of the operation of a Pascal compiler on a segment of code that invokes most of its major machinery?

For beginners the definition of *CONSTANT* is as much a puzzle as it is an example. Understanding it requires clear thinking and a careful reading of the code involved. This is in fact one of the major reasons for discussing it. FLINT's interpreter/compiler is put to work at three levels. At one level the word *CONSTANT* is being compiled. To understand the activity at this level, however, you must see through to the level at which *CONSTANT* will be executed. This level itself must be understood in terms of the activity that will occur at the next level—that is, when a particular instance of the word is executed. The activity at all of these levels is mediated by the same interpreter. The code defining its basic structure occupies half a page, and many of the supporting words—for example, *EXECUTE, COMPILE, HEADER, ":", CODE, ";"*, and *LITERAL*—are only two or three instructions long. Finally, if you stand back from the example a little, you see that an entire data type has been created by a nine-word statement. I feel that anyone who understands the simplicity underlying this example has a firm grasp of FLINT and should have no real trouble mastering Forth.

The Inner Shell

Once the embryonic FLINT system is running, it is ready for a big meal of nourishing words that will give it more size and power. These inner shell words support:

- definition and manipulation of constants, variables, and arrays
- stack manipulation and stack arithmetic
- structured control for branching and looping

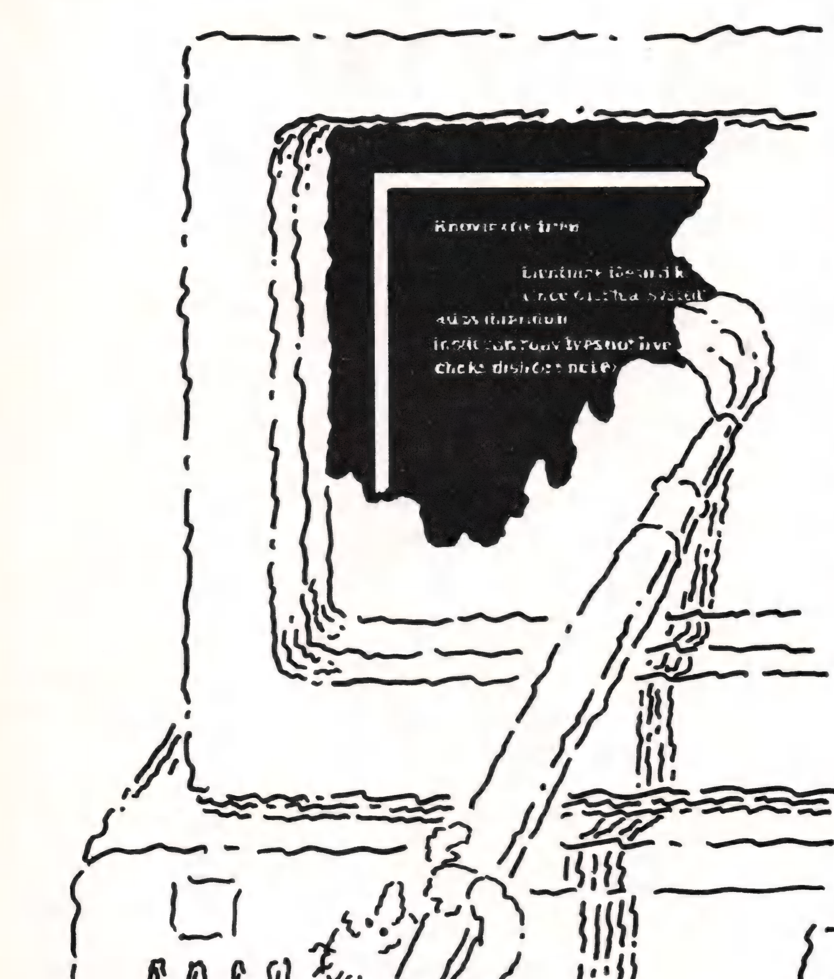
Listing Three, page 58, contains the inner shell words. Many of these words are defined directly in machine code (with the assembler mnemonics given as comments), so you might ask why they are not placed in the kernel and assembled as part of the basic system. My reasons for not doing so are primarily pedagogical. I wanted to maintain as much as possible the purity of the kernel to underscore its elegance and simplicity. In addition, the words, most of which are very short, seem to me to be more readable in Forth format than they are in assembly-language format. This representation, even allowing for comments, is much more compact because the work of building the headers is left to the system.

Structured control in Forth is directed by immediate words, placed in colon definitions to effect the compilation of appropriate conditional branching instructions. The flow of control is conditioned by values on the stack that act as flags. A zero value means false, and any nonzero value means true. As an example let's define the word *ODD*, which takes a number from the stack and prints an asterisk if it is odd:

```
: ODD 2 MOD IF 42 EMIT THEN ;
```

The action of *IF* when *ODD* is executed is to pop a value (assumed by *IF* to

TRY TO FIND A MORE VERSATILE **WINDOW** DEVELOPMENT TOOL.



Others have tried. Finally, a WINDOW tool for the serious C developer. Introducing ASPEN SCIENTIFIC'S CURSES WINDOW DEVELOPMENT PACKAGE.

- □ UNIX System 5 compatible function library
- □ Choice of Microsoft, Lattice, Computer Innovations or DeSmet compilers
- □ IBM EGA, CGA and monochrome displays as well as special modes for IBM BIOS and Microsoft ANSI driver compatibles (text modes only)
- □ MS-WINDOWS compatible using BIOS mode
- □ Full featured keyboard support
- □ Lightning fast, flicker free memory map output for IBM PC/XT/AT and compatibles
- □ CURSES library can be ported to the compiler and system of your choice
- □ Over 115 fully documented functions and macros
- □ Includes auto wordwrap, full color and special extension routines for PCs

What can all these features do for you? CURSES checks display types at run time so your source code does not have to be modified to work with different displays.* And with BIOS and ANSI support, your screen interface can run on virtually any PC using MS-DOS 2.0 or later. CURSES will enhance your product's human interface and save you time and money!

Let Aspen Scientific's CURSES decorate your product.

Call and order NOW, only 1000 copies will be sold at the low price of **$89.00.**

Requires DOS 2.0 or later, 192K, 1 Disk Drive.
*With use of run-time video attribute macro directives.

ASPEN SCIENTIFIC
P.O. BOX 72 WHEAT RIDGE,
COLORADO 80034-0072 (303) 423-8088

YES! I want my product to come alive with:

____ copies of CURSES Window Development Package $ 89 ea.
____ copies of CURSES including source code $289 ea.
Amount Enclosed
Price includes shipping to all U.S. cities $________
Outside USA make payment by International Postal Money Order

Payment: □ Check □ Money Order

My compiler is:

□ Microsoft C 4.0 □ Microsoft C 3.0 □ Lattice C 3.0
□ Computer Innovations C 86 Plus □ DeSmet C

Name ________
Company ________
Shipping Address ________
City ________ State ________ Zip ________
Telephone ________

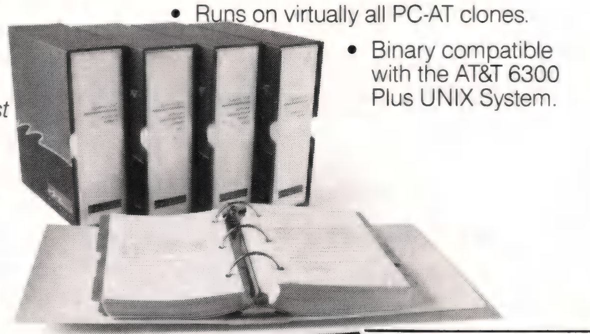
30 day money back guarantee.

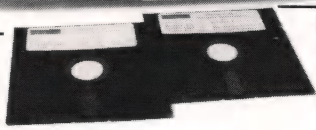
Circle **no. 360** on reader service card.

If you think you can't afford a UNIX® system. we've got a $160 surprise.

Turn your PC into a multi-user system.

Convert your IBM PC-AT (or compatible) into a multi-user/tasking UNIX work station—at *absolutely the best price* anywhere, any time. Based on the AT&T-certified UNIX System V/286, the MICROPORT SYSTEM V/AT is designed for use in virtually any computer environment, from office automation to software development.





Over 200 utilities come standard.

Grep, awk, sort, split, cut, paste, vi and ed (and many more) now let you search and modify files, make use of electronic mail, emulate terminals, calculate electronically, convert data and publish.

SYSTEM V/AT is more than a look-alike. It was derived from AT&T's own UNIX System V release 2 iAPX286. It thereby contains standard System V features the competitors don't support, such as the powerful symbolic debugger, sdb, the shell-layering job-control facility and the F77 Fortran compiler, as well as programming tools such as ctrace, cflow, and bs. Also standard is File System Hardening which greatly reduces data loss in a power failure.

Want some more features?

- Console driver providing ANSI terminal interface for monochrome, CGA, Hercules and EGA cards.
- Multiple Virtual consoles allow up to four virtual windows of operation.
- Record and File Locking
- Supports the 286's 16 megabyte virtual address space and fully utilizes its other advance features.
- Supports all standard IBM drive types and most non-standard hard-disk drives.
- Requires only one hard-disk partition, and allows DOS to reside on the same hard disk.
- Provides utilities to transfer files to-and-from DOS file systems.
- Dynamic disk buffer allocation provides RAM disk performance for systems with large memory configuration.
- Runs on virtually all PC-AT clones.
- Binary compatible with the AT&T 6300 Plus UNIX System.

Super software-development environment

We've provided everything: Make, yacc, lex, sccs, cflow, ctrace plus every standard System V software-development tool. The F77 Fortran compiler. And the AT&T Portable C compiler for the 286. Both C and Fortran compilers generate 287 instructions directly—for systems not containing 287 math coprocessors, a kernel-resident IEEE-compatible 287 emulator is provided. The large-model code produced by the compiler is among the densest and fastest currently available.

So, how do we do it?

MICROPORT offers SYSTEM V/AT at a fraction of the price of the competitors simply because we build on the generic System V/286 product from AT&T. This entire utility package from the certified release has been copied directly to SYSTEM V/AT—without so much as a recompile. Not only does this mean that MICROPORT can offer SYSTEM V/AT at a remarkable low price, it also *guarantees a level of quality* present in few (if any) other UNIX-system implementations. (And, since our staff was part of the group that implemented the standard System V/286 port for Intel, MICROPORT can offer comprehensive support for the system, as well.)

And a dollar change

The price is even better than you thought. Order right away and we'll return one silver dollar just as rapidly, with your product shipment. (If you'd like a little more time we'll apply that dollar to the cost of a brochure—which we'll send right away too.)

NOTE: Educational institutions—Call about our nominally-priced site licensing and source code

60 DAY MONEY-BACK GUARANTEE

And a dollar change.

To order: Complete the information below. Your attractively-packaged and fully-documented order will be shipped within two weeks.

MICROPORT SYSTEMS, INC.
4200 Scotts Valley Drive
Scotts Valley, CA 95066
408/438-UNIX or 800/PC2-UNIX (outside CA)

SYSTEM V/AT

- ☐ **RUNTIME SYSTEM** Includes the SYSTEM V/AT operation system and over 200 utilities, for two users. QUANT. ______ **$160.00**
- ☐ **SOFTWARE DEVELOPMENT SYSTEM** The complete Software Generation System for 286 development. QUANT. ______ **$169.00**
- ☐ **TEXT PREPARATION SYSTEM** Includes nroff, troff, spell and other programs. QUANT. ______ **$169.00**
- ☐ **THE COMPLETE SYSTEM** Contains all three packages indicated above. QUANT. ______ **$439.00**
- ☐ **OPTIONAL** three to eight-user upgrade. QUANT. ______ **$99.00**

Subtotal: ______

(CA residents add 6.5% tax per copy): ______

Shipping and handling charges (In the USA, $14.00; in Canada, $18.00; and in Europe, $110. . . . : ______

TOTAL DUE: ______

NAME ______

TELEPHONE ______

ADDRESS ______

CITY ______

STATE ______ ZIP ______

COUNTRY ______

☐ VISA ☐ MASTERCARD ☐ BANK DRAFT ☐ CHECK

CARD NUMBER ______ EXP DATE ______

☐ Send a brochure only and keep me on your mailing list, please.

M I C R O P O R T

Circle **no. 154** on reader service card.

68000 MINI FORTH

(continued from page 24)

be a flag) from the stack and test its value. If the value is false, execution continues at the word following *THEN* in the definition; otherwise, execution continues with the word following *IF*. In the above definition, the phrase *2 MOD* simply furnishes the proper flag for *IF* to "eat." The actions of *IF* and *THEN* when *ODD* is compiled (their compile-time behavior) must ensure that they have the desired effect when *ODD* is executed (execution-time behavior). Thus the definitions of *IF* and *THEN* will define their compile-time behavior, but they must be understood in terms of their execution-time behavior as well. An important point to remember here is that these words, as well as other control words, are not to be invoked directly; they act within definitions to achieve their effects.

Another control word, *ELSE*, can be used optionally with *IF* and *THEN*. If, for example, you wish to redefine the word *ODD* so that its action is to print the value of an odd number or else drop the number, you might try

```
: ODD DUP 2 MOD IF . ELSE \ THEN ;
```

The execution-time behavior of *ELSE* is, as you might suspect, to transfer control to the word following *THEN*. A subtle point here is that now *IF* must transfer control to the word following *ELSE* rather than to the one following *THEN* when *ELSE* is present.

FLINT provides two structured loop control mechanisms: *DO . . . UNTIL* and *DO . . . WHILE . . . LOOP*. The execution-time behavior of *UNTIL* is to eat a flag and pass control back to the word following *DO* if its value is true. Otherwise the loop completes when control passes to the word following *UNTIL*. *WHILE*, on the other hand, makes an exit to the word following *LOOP* if its value is false. *LOOP* always passes control to the word following *DO*. (Note that FLINT's usage of these words is different from standard Forth's; they seem to make a little more sense to me this way.)

Nowhere, I feel, is the elegance and power of Forth's programming paradigm more in evidence than in the definition of the lexicon of words supporting FLINT's structured control. The words *?>*, *BRA>*, *BEQ>*, and *BNE>* form the raw material for the tests and branches, and the words *MARK*, *SPLIT*, and *JOIN* provide the tools for assembling them. Each of the control words is then built economically by a single defining word or phrase, and yet when they are used any of the resulting control structures can be nested to arbitrary depth within any of the others!

Forth's elegance and power is evidenced by the lexicon of words supporting FLINT's structured control.

There remains the problem of entering these definitions interactively. In theory this is no worse than entering them into an assembly-language code file—except of course that all is lost when the system is turned off or (as is more likely) crashed by the novice user. My own approach to the problem was to try to find a way to save an image of the augmented system. As it turns out, all that is necessary for saving the state of the system is to preserve the contents of the internal dictionary pointer, register *A5*, which points to the next vacancy in the dictionary. (This happens to be the area used by *TOKEN* as temporary storage for the words that it extracts from the line buffer.) The word *LOGOFF* accomplishes this task by saving *A5* in the system variable *BUFPNT*, from which it is loaded each time *FLINT* runs. For this solution to work, you must have the means (via a monitor/debugger, for instance) to save and reload the augmented system in place of the original. Lacking such a tool, you should probably just bite the bullet and build the inner shell words into the source in the manner of the kernel words so that they will be assembled directly in the dictionary. This is tedious work, but it is not difficult if you understand the dictionary structure.

Ultimately, of course, it would be nice to dispense with these primitive methods and be able to create and edit disk-resident files of FLINT source text. In fact, the words *SAVE*, *LOAD*, and *GO* are included in the kernel system to support this very activity. The word *GO* directs the outer interpreter to take its tokens from a 1K reserved area known as the disk/edit buffer instead of from the line buffer. After the outer interpreter has exhausted the contents of this buffer, it will encounter the token *INTERACTIVE*, which has been placed in memory just beyond the buffer area. Execution of this word reinitiates the normal interactive input sequence and redirects the outer interpreter to take tokens from the line buffer. *LOAD* and *SAVE* allow the contents of this area to be read from or written to an absolute disk block called the current block, which is specified as the value of the system variable *CBLOCK*. A description of these words begs the obvious question of how usable text gets into the buffer (or on disk) in the first place.

My own solution was to use the expanded vocabulary of the inner shell to write a simple editor. Once this editor was built interactively, it could be used to place the text for the inner shell (as well as itself) in the edit buffer for eventual storage on disk via *SAVE*. These blocks of text are called screens. Careful examination of FLINT's initialization procedure will show that it loads and executes screen #80 (the initialized value of *CBLOCK*). This block and the succeeding ones are where the text for the inner shell and the editor have been placed. The word -> (defined on screen #80) directs the interpreter to increment *CBLOCK* and load and execute the next text screen. It allows an entire string of screens to be executed without interactive direction. By this device the inner shell and editor are effectively booted by FLINT whenever it is invoked. The final result is a fairly mature system that contains most of the basic features of a true Forth and enough legitimate tools to be quickly expanded further.

You can characterize the basic outline of FLINT's construction as a bootstrap procedure, which is really just

an accelerated and miniaturized version of the conventional process whereby low-level tools are used to build higher-level ones. The assembler is used to build an interactive interpreter/compiler (the FLINT kernel), which is used to incorporate a more sophisticated set of tools (the inner shell). These are then used to build an editor that allows the inner shell, the editor, and all future extensions to become permanent parts of the system.

The Aftermath

What, you might ask, did the teacher learn from all this? Well, for one thing, I relearned the value of ignorance. I sailed pretty far into the project on the momentum of my initial enthusiasm. It lasted, in fact, until everything was pretty much complete and working. Unfortunately, the job is not done when the program works. Cleaning up little messes, pruning, tuning, documenting, testing, and tracking down the subtle bugs—in short, the real work—was equally time-consuming but much less rewarding. Determination and good old-fashioned stubbornness had to finish what enthusiasm had begun. If I had fully realized in advance the amount of extra work involved, I would probably never have started. The moral here is that underestimating the magnitude of a task is often a necessary condition for attempting it. The work is still not finished, of course. It never is. The stack is in the wrong place, the realization of the *CODE* submode is imperfect, and so on, and so on. Any program is only an approximation of what it should be.

As far as Forth itself is concerned, there are several things to be said. I have acquired a terrific amount of respect for the ingenuity of Forth's inventor, Charles H. Moore. *Work of genius* is a term properly applied to such an engagingly simple yet immensely practical conception. I have not yet become, however, one of Forth's true believers. Programming is not my profession—it's actually more of a hobby—so I don't have enough real knowledge or experience to make any credible claims for its superiority or inferiority to other programming languages. I'll leave that issue to the people who make their money writing programs. My opinion is that Forth's programming paradigm of building the language to fit the problem allows (and indeed requires) a much more flexible and imaginative approach to programming problems than is called for with many more traditional languages.

The extra degree of freedom actually seems to require more self-discipline on the part of the programmer to use it wisely, but it makes problem solving more fun. I find that my programs tend to become compositions that I judge on aesthetic as well as functional grounds.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063 or call (415) 366-3600 ext. 216. Please specify the issue number and disk format (MS-DOS, Macintosh, Kaypro).

DDJ

(Listings begin on page 52.)

Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

The OS-9 Operating System

by Brian Capouch

The OS-9 operating system is a modular, multiprogramming, multitasking operating system (OS) that runs on a large variety of Motorola microprocessors. Designed to be conceptually similar to Unix, it provides programmers with a good environment for the same reasons that Unix does: it uses a hierarchical file structure, allows command-line invocation of concurrent processes, has a similar implementation of pipes and filters, and permits device-independent I/O. The differences are more important than the similarities, though.

Mean and Lean

OS-9 differs from Unix in several significant respects. It is, to turn a phrase, "meaner and leaner" than Unix. It occupies 12K of address space in its smallest 6809 incarnations and a little more than 48K in its full-blown 68020 form. It is more dynamically changeable than Unix, allowing users to add I/O devices and update system modules without rebooting a running system. It was also designed specifically to accommodate ROMed software easily and, for a variety of reasons, has proven suitable for real-time and process-control applications. These and other features have assured OS-9 a niche encompassing a wide variety of markets. This article gives an overview of the design of OS-9, particularly focusing on those aspects that make it special. I've included a pair of small application programs that are intended to give you the flavor of the OS-9

OS-9 is meaner and leaner than Unix.

programming environment

Origins

OS-9 originally had its genesis as part of a contract project between its designer, Microware Systems Corp., and Motorola, during the time that the hardware design of the 6809 processor was being finalized. The original concept of the project was to provide a modern, structured version of the BASIC programming language that would take advantage of the features of the 6809. The resultant language, called Basic09, incorporates several features that were at the forefront of language design; I'll review it briefly with examples later.

As the language development proceeded, Microware, with an eye toward the developing academic and commercial use of Unix, began developing an operating environment to complement Basic09's structure and modularity. Thus OS-9 was born. The language and the operating system appeared together in 1981, a few months after the 6809 came into production. As the design for the 68000 was begun shortly thereafter, Microware began a port of OS-9 for that processor.

As the 68000 and its successors have appeared in a multitude of applications, OS-9 has been ported into hundreds of designs. OS-9 currently appears on a wide range of machines, ranging from the 6809-based Tandy Color Computer to the GMX Micro-20, a 68020-based system that provides 2-megabytes of RAM, SASI and floppy-disk interfaces, serial and parallel I/O ports, and a 68881 math coprocessor on a card that mounts on a 5¼-inch disk drive. This machine has a performance that outstrips a variety of machines, including the VAX 11/780.

Named Modules

Structurally speaking, perhaps the most significant feature of OS-9 is its extreme level of modularity (see Figure 1, page 31). The underlying model of the OS-9 address space is a dynamic collection of named modules. When an executable module comes to life as a process, it is associated with a separate area of memory that is used to store its data. All code exists in what are known as memory modules, whether located in primary or secondary store. A memory module, generally, is a segment of code sandwiched between a header and a cyclic rendundancy checksum. The header contains information about the module that is used by the OS both during the time that the module is resident in memory and during transfers to and from secondary storage devices. The checksum is used to verify the integrity of the module during transfers to and from secondary store.

Modules that contain object code must consist of position-independent code and are shared automatically between various users of the system. This avoids having copies of the same object code occupying multiple sections of memory. In such cases, the OS assigns each user of the module to a different area of memory for data storage.

Brian Capouch, R. R. 2 Box 151, Monon, IN 47959-9229. Brian is a self-taught computer scientist who teaches at St. Joseph's College in Rensselaer, Ind.

The first bytes of each module are "sync bytes." These bytes, which consist of unused machine opcodes, are used at start-up to locate and load ROMed modules automatically. OS-9 permits ROMed code to be updated dynamically, simply by loading into memory a module with the same name as the ROMed module but with a higher revision number in its header. The mixture of modules that constitutes the "current environment" is more dynamic than in most other OSs, a feature that has opened up many application avenues.

Layered Hierarchy

Like Unix, OS-9 is organized as a layered hierarchy of functional modules. Each component is a memory module, as described earlier. At the lowest level of the hierarchy is the kernel, aided by small init and clock modules as well as several others. The OS-9 kernel handles all basic OS functions, including process scheduling and dispatch, memory management, and basic I/O processing. It determines from the init module the exact characteristics of its environment, which under OS-9 more than under Unix is likely to differ from one system to another. The clock

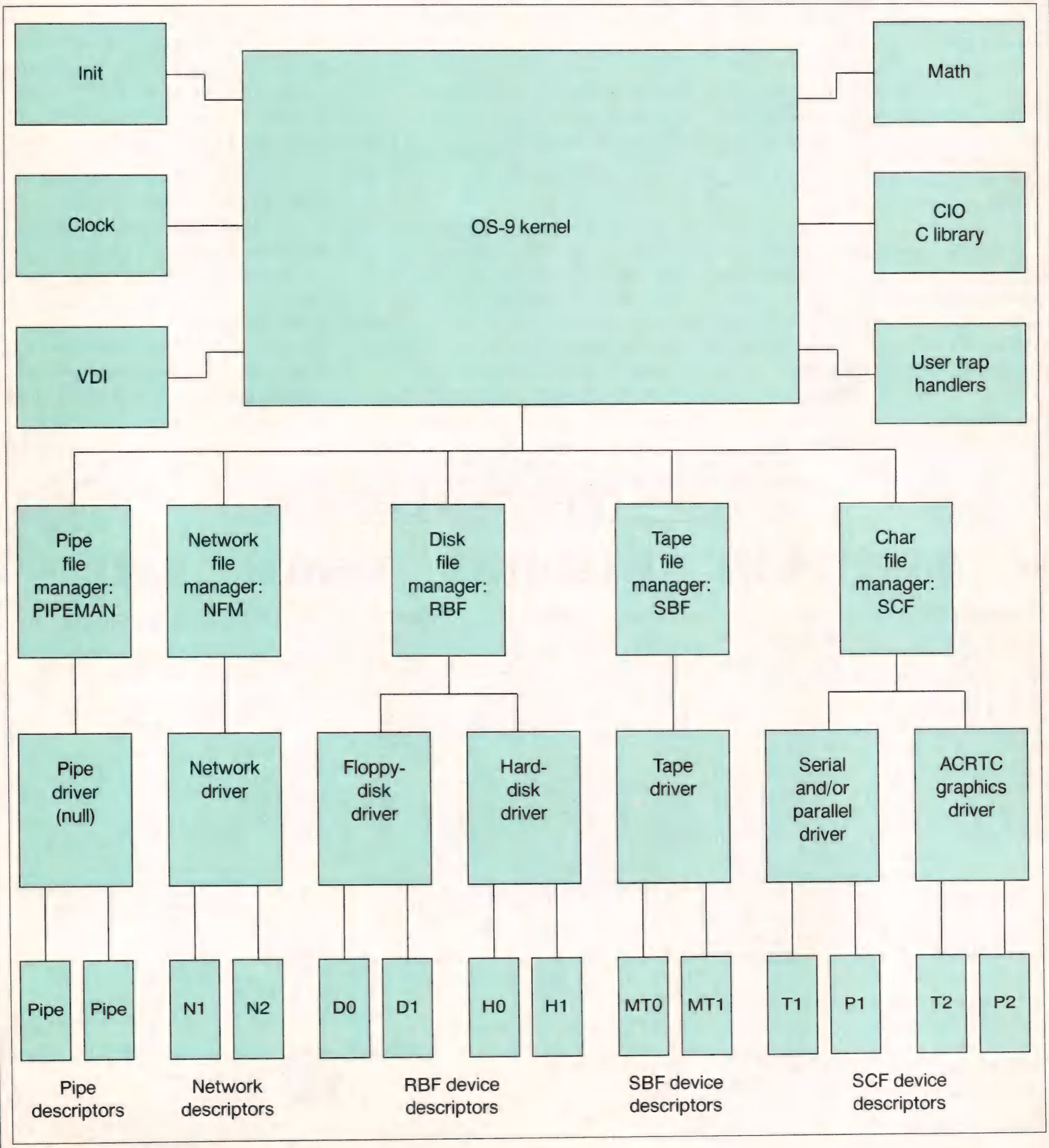


***Figure 1:** OS-9 module organization*

OS-9 OPERATING SYSTEM
(continued from page 31)

module interfaces to a tick generator for time-slice timing.

At the next layer outward from the basic system modules lie the file manager modules. Each of these is designed to interface the I/O data stream into and out of the processor and a particular class of similar devices. Because the conditioning is performed outside the kernel, all data appears to the CPU as equivalent byte streams. The module RBF interfaces to random-block-oriented devices, such as disks. SCF handles character streams that are bound for both parallel and serial I/O devices, and PIPEMAN coordinates data streams between concurrently running processes.

Each file manager conditions its data stream and passes it on to a device driver module. A device driver must exist for each different type of hardware to which the system is interfaced. The driver, such as a disk controller or intelligent interface processor, may control a large number of devices, and it is this level of the system that hides hardware dependencies from the lower levels of the OS.

At the highest level of abstraction are the device descriptor modules. The system needs one of these for each individual I/O device. They contain device-dependent parameters, such as port addresses, disk blocking factors, control characters, and so on.

The OS-9 unified I/O system is dynamically configurable, making it quite different from Unix. Devices can be added to and removed from running systems on the fly, and through changing device descriptors, different devices can be added to the same port and referenced interchangeably.

The 68xxx versions of OS-9 contain a complete math subroutines library that handles a wide variety of floating-point, extended-precision integer, and transcendental math functions. This package is called via the *Trap* instruction. Programs that use the library can be used without change on systems that implement hardware coprocessors by changing the trap handler associated with the call.

The User Interface

The OS-9 user communicates with the OS through a user interface called shell. Shell is similar to Unix shells, although it differs in enough respects to wreak havoc with the neuronal patterns of folks who try to time share between the two different OSs. Many of the utility commands go by what Microware thinks are saner names than their Unix equivalents, and utilities that depend on file structure and process specifics are of necessity organized differently.

One notable difference between the two shells is a set of OS-9 control characters that speed up line editing at the shell level. For instance, in other environments I have found myself constantly wishing for a "repeat line" control character (commonly assigned to Control-A) that provides the function "reenter the contents of the line input buffer up to the most recent carriage return entered." High-speed

Programmer's Paradise Gives You Superb Selection, Personal Service and Unbeatable Prices!

Welcome to Paradise. The PC/MS-DOS software source that caters to your individual programming needs.

Discover the Many Advantages of Paradise...

- Lowest price guaranteed
- Latest versions
- Huge inventory
- Immediate shipment
- Special orders
- 30-day money-back guarantee

We'll Match Any Nationally Advertised Price.

	LIST	OURS
C++		
ADVANTAGE C++	$ 495	CALL
PFORCE++	395	CALL
C COMPILERS		
C-86 PLUS	497	CALL
DATALIGHT – C	60	49
DATALIGHT – C DEVELOPER'S KIT	99	79
LATTICE C 3.2	500	289
LATTIC C W/SOURCE	900	545
LET'S C	75	59
W/CSD DEBUGGER	150	109
MICROSOFT C 4.0	450	285
MARK WILLIAMS C	495	289
SUPERSOFT C	395	339
WIZARD C	450	369
C INTERPRETERS		
C-TERP	300	235
INSTANT C	500	379
INTRODUCING C	125	105
RUN/C	150	89
RUN/C PROFESSIONAL 1.1	250	169
ASSEMBLERS, LINKERS		
386|ASM	495	395
ADVANTAGE LINK	495	CALL
MACRO-86	150	98
PASM-86	195	135
PLINK 86 PLUS	495	335
QUELO 68000 X-ASM	595	509

January Specials From Phoenix

PASM-86	195	135
PDISK	195	135
PFANTASY PACK	1295	889
PFINISH	395	245
PFIX PLUS	395	245
PFORCE	395	245
PLINK 86 PLUS	495	335
PMAKER	125	95
PMATE	195	125
PRE-C	295	165

GRAPHICS		
ESSENTIAL GRAPHICS	250	205
GSS GRAPHICS DEVELOPMENT TOOLKIT	495	389
GSS KERNEL SYSTEM	495	389
GSS METAFILE INTERPRETER	295	239
GSS PLOTTING SYSTEM	495	389
HALO—ONE LANGUAGE	300	209
HALO—FIVE MICROSOFT LANGUAGES	595	415
METAWINDOWS	185	115
METAWINDOWS PLUS	235	189
METAFONTS	80	70
METAFONTS PLUS	235	189

	LIST	OURS
C UTILITY LIBRARIES		
ASYNC MANAGER	175	135
BASIC C	175	129
C ESSENTIALS	100	85
C FOOD SMORGASBORD	150	98
W/SOURCE	300	188
C TOOLS PLUS	175	135
ENTELEKON COMBO PACKAGE	200	169
C FUNCTIONS LIBRARY	130	109
C WINDOWS	130	109
SUPERFONTS FOR C	50	43
ESSENTIAL C UTILITY LIBRARY	185	135
ESSENTIAL COMM LIBRARY	185	135
W/BREAKOUT DEBUGGER	250	195
GREENLEAF FUNCTIONS	185	135
GREENLEAF COMM	185	135
THE HAMMER	195	175
MULTI C	149	135
PFORCE	395	245
TIMESLICER	295	265
TOPVIEW TOOLBASKET	250	189
SCREEN DISPLAY, WINDOWS		
C WORTHY	295	269
CURSES	125	94
W/SOURCE	250	184
FLASH UP WINDOWS	75	68
MICROSOFT WINDOWS DEVELOPMENT SYSTEM	500	329
ON-LINE HELP	149	109
PANEL	295	224
SCREENPLAY (LATTICE)	150	135
SOFTSCREEN HELP	195	175
VIEW MANAGER	275	199
VITAMIN C	150	135
VC SCREEN	99	84
WINDOWS FOR C	195	145
WINDOWS FOR DATA	295	250
Z VIEW	245	189
FILE MANAGEMENT		
BTRIEVE	245	195
XTRIEVE	245	195
W/REPORT GENERATION	390	315
BTRIEVE/N	595	465
XTRIEVE/N	595	465
W/REPORT GENERATION	940	750
C TREE	395	329
R TREE	295	265
CQL	395	329
DBC III	250	189
W/SOURCE	500	379
DB VISTA	195	155
W/SOURCE	495	425
DB QUERY	195	155
W/SOURCE	495	425
FABS	150	129
FABS PLUS	195	169
INFORMIX	795	639
INFORMIX 4GL	995	799
INFORMIX SQL	795	639
PHACT	295	265

	LIST	OURS
SORT UTILITIES		
AUTOSORT	150	129
M/SORT	155	139
OPT-TECH SORT	149	115
MAKE, LINT, PROFILE, UTILITIES		
C CROSS REFERENCE GENERATOR	50	39
LMK	195	145
POLYMAKE	99	78
OTHER POLYTRON PRODUCTS	CALL	CALL
PMAKER	125	95
PFINISH	395	245
THE PROFILER	125	94
PC LINT	139	105
PRE-C	295	165
TEXT MANAGEMENT UTILITIES	120	94
DEBUGGERS		
ADVANCED TRACE 86	175	139
BREAKOUT	125	99
CODESMITH 86	145	108
C SPRITE	175	138
CI PROBE	75	59
CSD SOURCE DEBUGGER	75	59
PERISCOPE I	295	249
PERISCOPE II	145	109
PERISCOPE II-X	115	85
PFIX 86 PLUS	395	245
XVIEW 86	60	49

Featured Product of the Month

ADVANTAGE LINK—the first overlay linker to take advantage of extended memory. Link object modules from Microsoft languages, RM COBOL and Fortran. Lattice C, and many more. Supports memory caching, object file merging, complex overlay structures and automatic overlay reloading.
LIST $495 **Intro Special $349**

EDITORS		
BRIEF	195	CALL
CVUE	75	59
W/SOURCE	250	195
EDIX	195	155
EMACS	295	265
EPSILON	195	159
FIRSTIME (C)	295	229
KEDIT	125	105
LSE	125	95
PMATE	195	125
PC/Vi	149	129
SPF/PC	195	149
VEDIT	150	109
VEDIT PLUS	225	139

	LIST	OURS
PASCAL COMPILERS		
MICROSOFT PASCAL	300	189
PASCAL 2	395	355
TURBO PASCAL	100	69
OTHER BORLAND PRODUCTS	CALL	CALL
TOOLS FOR TURBO PASCAL		
ALICE	95	68
FIRSTIME	75	59
FLASH UP WINDOWS	75	68
HALO	300	209
SCREENPLAY	100	89
SCREEN SCULPTOR	125	94
T-DEBUG PLUS	60	50
TURBO EXTENDS	85	65
TURBO PASCAL ASYNC MGR	100	84
TURBO PROFESSIONAL	70	49
TURBO POWER TOOLS PLUS	100	83
TURBO WINDOWS	80	65

NEW Products

386|ASM/LINK—Complete development package for 80386 microprocessor including an assembler, linker, and debugger. Upwardly compatible with Microsoft's Macro Assembler.
List $495 Ours **CALL**

LATTICE C—Version 3.2—Features full support for Microsoft Windows including the "Far," "Near," and "Pascal" key words.
List $500 Ours **$289**

PASCAL 2—Highly optimized Pascal compiler, with source level debugger, profiler.
List $395 Ours **$355**

PFORCE++—Huge library of functions designed specifically for object-oriented programming with C++.
List $395 Ours **CALL**

RUN/C PROFESSIONAL—Version 1.1—Now compatible with Microsoft 4.0! Loadable libraries advanced debugging features.
List $250 Ours **$169**

TIMESLICER—Multitasking, linkable library supporting concurrent tasks and real-time event processing with header files provided for both C and Assembly.
List $295 Ours **$265**

BASIC		
BETTERBASIC	199	139
SUMMIT ADD ONS	CALL	CALL
BETTER TOOLS	95	89
FINALLY	99	89
MICROSOFT QUICKBASIC	99	75
PROFESSIONAL BASIC	99	75
8087 MATH SUPPORT	50	45
PANEL-BASIC	145	115
RM/BASIC	600	479
TRUE BASIC	150	105

OTHER PRODUCTS AVAILABLE TO THE BASIC PROGRAMMER INCLUDE MULTIHALO, BTRIEVE, GSS GRAPHICS, SCREEN SCULPTOR, STRUBAS, 87 BASIC.

COBOL COMPILERS/UTILITIES		
MICROSOFT COBOL	700	445
MICROSOFT COBOL TOOLS	350	205
MICROSOFT SORT	195	139
MICRO/SPF	175	CALL
OPT-TECH SORT	149	115
REALIA COBOL	995	785
SCREENPLAY	175	155
RM/COBOL	950	639
RM/COBOL 8X	1250	895
VISUAL COBOL (MBP)	1150	1015
FORTRAN UTILITIES		
ACS TIMES SERIES	495	419
87 SFL	250	225
FOR-WINDS	90	78
FORLIB-PLUS	70	54
GRAFMATICS OR PLOTMATICS	135	119
GRAFMATICS AND PLOTMATICS	240	219
FORTRAN SCIENTIFIC SUBROUTINES	295	249
POLYFORTRAN TOOLS I	179	143
STRINGS AND THINGS	70	54

ALSO AVAILABLE TO THE FORTRAN PROGRAMMER: PANEL, MULTIHALO, BTRIEVE, ESSENTIAL GRAPHICS, FLASH UP WINDOWS, GSS GRAPHICS, OPT-TECH SORT.

PROLOG		
ARITY PROLOG (STANDARD)	95	59
ADDIT. ARITY PRODUCTS	CALL	CALL
CHALCEDONY PROLOG	100	89
TURBO PROLOG	100	79

LISP, OTHER AI, CALL FOR INFORMATION, PRICING, AVAILABILITY.

TRANSLATORS/BRIDGES		
BASTOC (MBASIC)	495	399
C TO DBASE	150	135
DBC III	250	189
W/SOURCE	500	378
D BX	350	329
FORTRIX	6000	CALL
R:BRIDGE	395	319

Terms and Policies

- **We honor MC, VISA, AMERICAN EXPRESS**

No surcharge on credit card or C.O.D. Prepayment by check. New York State residents add applicable sales tax. Shipping and handling $3.00 per item, sent UPS ground. Rush service available, prevailing rates.

- Programmer's Paradise will match any current nationally advertised price for the products listed in this ad.
- **Mention this ad when ordering**—some items are specially priced.
- Prices and Policies subject to change without notice.
- Corporate and Dealer inquiries welcome.

1-800-445-7899 In NY: 1-800-642-6471

Programmer's Paradise
487 E. Main Street, Mt. Kisco, NY 10549
914-332-4548

Programmer's Paradise

Circle **no. 334** on reader service card.

OS-9 OPERATING SYSTEM
(continued from page 32)

interaction with any processor leads to a large number of typographical mistakes, such as misspelling directory, file, or command names. The repeat-line character allows for quick reentry of the offending line, then backspacing to the point of error without having to retype the preceding characters. Although this may seem a picayune advantage to some, it can become addictive. Other control characters allow canceling the line, pausing the screen display, interrupting and canceling the currently running program, and so on. The characters that these functions map to are defined in the device descriptor and can be changed with the utility programs tmode and xmode. OS-9 also supports Unix-style type-ahead on serial input devices.

Development Tools

OS-9 programmers can choose from a variety of programming languages and functional processors. The OS-9 C compiler is Unix compatible down to the standard library level, and most of the system calls accessible from C have been assigned names that correspond to their Unix equivalents. One OS-9 hacker reported developing a reasonably complex program that included several low-level I/O calls using OS-9 C on his Radio Shack Color Computer. He then ported the program to a VAX 11/780, where it compiled and ran without a single change.

Besides C, there also exist compilers for Pascal, FORTRAN-77, COBOL, and at least four different versions of the BASIC language. These include Basic09, the language for which OS-9 was originally developed. Basic09 is an interesting and unusual language, and the example programs presented later demonstrate some of its features. Besides language processors, there is the usual plethora of functional processors, such as text editors, formatters, and so on. The standard OS-9 assembler is a relocatable macro assembler that allows management of complex assembly-language code in a variety of libraries that can be assembled separately.

More Differences

A few sundry points also underscore the differences between OS-9 and

```
Microware OS-9/68000 Resident Macro Assembler V1.6  86/08/18  14:02  Page     1
 syscall.a
OS-9 Assembly constants -
00001 *! Syscall Routine - From Microware Manual
00002                          use      <oskdefs.d>
00001                          opt      -l
00072
00003
00004  00000000                org      0
00005  00000000 Return         do.l     1
00006  00000004 Length1        do.l     1
00007  00000008 Param2         do.l     1
00008  0000000c Length2        do.l     1
00009
00010                          psect    SysCall,(Sbrtn<<8)!Objct,(ReEnt<<8)!1,0,0,SysCall
00011
00012 0000 0c80 SysCall        cmpi.l   #2,d0           check parameter count
00013 0006 6640                bne.s    ParamErr        branch if error
00014 0008 0caf                cmpi.l   #4,Length1(a7)  is first parameter integer?
00015 0010 6636                bne.s    ParamErr        branch if not
00016 0012 0caf                cmpi.l   #52,Length2(a7) 52 bytes of registers?
00017 001a 652c                blo.s    ParamErr        branch if not
00018 *!        Now put model on the stack
00019 001c 343c                move.w   #Modllen/2,d2   number of words for dbra
00020 0020 41fa                lea      Model+Modllen(pc),a0 get address of model
00021 0024 6002                bra.s    SysCO2          branch into loop
00022 0026 3f20 SysCO1         move.w   -(a0),-(a7)     move a word
00023 0028 51ca SysCO2         dbra     d2,SysCO1       continue if not done
00024 002c 2041                move.l   d1,a0           point to function code
00025 002e 3f68                move.w   2(a0),2(a7)     set function code
00026 *!           Get the registers
00027 0034 2a6f                movea.l  Param2+Modllen(a7),a5 get address of parameter
00028 0038 4cdd                movem.l  (a5)+,d0-d7/a0-a4 get register
00029 003c 4e97                jsr      (a7)            call function
00030 003e 48e5                movem.l  d0-d7/a0-a4,-(a5) copy register set
00031 0042 4fef                lea.l    Modllen(a7),a7 clear stack
00032 0046 4e75                rts
00033
00034 0048=323c ParamErr       move.w   #E$Param,d1     get error code
00035 004c=003c                ori      #Carry,ccr      set carry
00036 0050 4e75                rts
00037
00038 0052=4e40 Model          os9      F$Fork          model system call to put on stack
00039 0056 4e75                rts
00040
00041  00000006 Modllen        equ      *-Model
00042  00000058                ends
00043
Errors: 00000
Memory used: 19k
Elapsed time: 2 second(s)
```

Code Example 1: *Syscall, an OS-9 resident macro assembler*

Unix. First, the OS-9 kernel is written in assembly language instead of in C. Although this restricts the OS to a small set of machines, it results in faster, more compact code. OS-9 also uses a different scheduling algorithm—one that results in noticeably faster throughput than most Unix implementations. Because of its modular memory management and dynamic configurability, OS-9 lends itself better to low-level control applications and real-time processing. On the other hand, OS-9 does not swap programs into and out of primary store. Although this speeds up throughput considerably, it also means that once the total available RAM in a system is occupied, no more jobs can be run. OS-9 has no limit on the number of concurrently executing processes—one user reports spawning more than 600 of them on his 68020 system before running out of memory. OS-9 users also report that the substantially smaller memory requirements and faster speed of OS-9 suit it for many applications in which Unix is too large and slow.

Basic09

The example programs I have included with this article are designed to illustrate some interesting facets of OS-9. The Basic09 programming language is a highly modular implementation of BASIC that is not only compatible with most sophisticated versions of that language but also contains many structures that are seen in Pascal. It is an interpretive compiler that attempts to balance the better points of both translation processes. The language contains both an integral editor and debugger. Source code is compiled, a line at a time as it is entered, into an intermediate code. Syntax errors are reported to the user as each line is entered and can be immediately corrected using the editor. Programs consist of any number of named procedures in which a variety of data types can be declared and built up into named Pascal-record-like structures. Parameters can be passed between procedures by either value or reference, and *TRACE*, *PAUSE*, and *STATE* statements allow for interactive debugging.

The example programs illustrate some specific features: first, Basic09 programs can call and pass parameters to and from programs written in assembly language or compiled by other compilers. The program Syscall (Code Example 1, page 34) is a 68000 program that allows a Basic09 program to perform system calls from within a procedure simply by passing a 68000 register image and an OS function code. Because it is reasonably closely tied to the calling syntax for OS-9 assembly-language system calls and outside the scope of this article, I have chosen not to discuss it here and simply treat it as a black box. Procedure *ShowCall* (Code Example 2, below) illustrates how Syscall would be invoked in a Basic09 procedure. In this case, I simply use the Syscall routine as a substitute for the high-level Basic09 *OPEN* command. The system call, which opens the file for read, passes back a path number in the *d0* register, which is then assigned to the Basic09 variable *Path*.

The second example program, FormLetter (Code Example 3, page 36), illustrates a Basic09 procedure that adds a front end onto a standard text formatter. In order to personalize a form letter that I was preparing for a basketball team I coach, I wanted to incorporate names, addresses, and greetings from a demographic file into the body of a standard letter. My local implementation of the formatter does not permit reading from outboard files into the standard data stream. I chose to put together tools that I already had at hand rather than do a "Swiss army knife" modification to the text formatter. (At this point, please realize that I am well aware of the case for making a permanent addition to the formatter but would have lost an example for this article.)

The technique I chose was to create a pair of named pipes, one of which receives a stream of data headed for the formatter, the other one receiving its output. I then connected those pipes to file paths within the Basic09 procedure via calls to the OS-9 shell. It was possible thereby to integrate my demographics with the body of my letter, sending the merged data as standard input to the formatter. In this example I could then send the output of the formatter to a file, but I could have just as easily invoked yet another pipeline to have it processed directly by my system's print spooler. (Basic09 commands to do so are enclosed as comments within the pro-

```
PROCEDURE ShowCall
 0000
 0001      (* Procedure to demonstrate System Call from Basic09
 0036      (* Define complex data type to represent registers
 0069      TYPE registers=d0,d1,d2,d3,d4,d5,d6,d7,a0,a1,a2,a3:
                                                        INTEGER
 00A0
 00A1      (* Declare necessary variables
 00C0      DIM Path:INTEGER
 00C7      DIM Line:STRING[128]
 00D3      DIM IOPEN:INTEGER
 00DA      DIM Regs:registers
 00E3      DIM FileName:STRING[48]
 00EF
 00F0      (* Initialize Variables
 0108      (* IOPEN is OS-9 System Call I$OPEN: Open path to a file
 0141      IOPEN=$84
 0149      (* This is the file we'll be reading from
 0173      FileName="BOpen"
 017F
 0180      (* Set up register image for system call
 01A9      Regs.d0=1
 01B4      Regs.a0=ADDR(FileName)
 01C2      RUN Syscall(IOPEN,Regs)
 01D1
 01D2      (* Assign returned path number to Basic09 variable
 0205      Path=Regs.d0
 0210
 0211      (* Read and print file contents
 0231      WHILE NOT(EOF(#Path)) DO
 023C        READ #Path,Line
 0246        PRINT Line
 024B      ENDWHILE
 024F
 0250      (* Use Basic09 interface to close file
 0277      CLOSE #Path
 027D      END
```

Code Example 2: *Procedure to invoke Syscall*

OS-9 OPERATING SYSTEM

(continued from page 35)

gram.) These concepts can be extended in vastly more complex ways, all the while yielding the benefits of totally modular construction and interactive program development. The interactivity of Basic09 has left me consistently choosing it rather than C when portability is not a requisite, and I have found development to be much faster because of the interactive compilation, editing, and debugging.

Conclusions

I am tempted to call OS-9 the "poor man's Unix," even though I have to marvel at the power of its most sophisticated implementations, which are on par (or superior) to Unix running on a VAX in terms of speed and memory utilization. The concept of named, automatically located memory modules is an innovation that extends its capabilities into process control and real-time applications that are unsuitable for Unix, and its user interface and overall organization enable those familiar with Unix to adapt to its operation quickly. Its growing user base indicates that what was considered for many years an underground classic is now emerging from the shadows, and it should provide an interesting and productive environment for programmers of machines based on Motorola processors for years to come.

I would like to thank GMX Corp. for providing me with one of its Micro-20 systems for evaluating that implementation of OS-9.

Availability

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's Journal*, 501 Galveston Dr., Redwood City, CA 94063 or call (415) 366-3600 ext. 216. Please specify the issue number and format (MS-DOS, Macintosh, Kaypro).

DDJ


```
PROCEDURE FormLetter
 0001       (* Formletter: Add a front-end to K & P text formatter
 0039       (* Written by Brian Capouch, 7/86
 005C       (* Define complex type to hold demographic info
 008C       TYPE NameRec=Name,Add1,Add2,Greeting:STRING[80]
 00A8
 00A9       (* Declare variable storage
 00C5       DIM Record:NameRec
 00CE       DIM NameFile,BodyFile:STRING[48]
 00DE       DIM Includer:STRING[10]
 00EA       DIM NamPath,SplPipe,InPipe,OutPipe:INTEGER
 00FD       DIM Counter:INTEGER
 0104       DIM Line:STRING[256]
 0110       DIM Index:INTEGER
 0117
 0118       (* Trap out EOF coming down pipeline
 013D       ON ERROR GOTO 10
 0143
 0144       (* Initialize variables
 015C       NameFile="Demographics"
 016F       Includer=".us Body"
 017E
 017F       (* Set up files
 0190       OPEN #NamPath,NameFile:READ
 019C       GET #NamPath,Counter
 01A6
 01A7       (* Set up output path to file
 01C4       CREATE #SplPipe,"LettersOut":WRITE
 01D9
 01DA       (* Send each set to formatter, with preface and suffix
 021E       FOR Index=1 TO Counter
 0230         (* Set up pipes to text formatter
 0252         (* This procedure "owns" this named pipe
 027B         CREATE #InPipe,"/pipe/in":WRITE
 028E         (* Now couple this pipe to text formatter
 02B8         SHELL "runb tformat </pipe/in >/pipe/out&"
 02DF         (* Open path to output pipe from formatter
 030A         OPEN #OutPipe,"/pipe/out":READ
 031F         (* Next two lines commented out
 033E         (* They would send output to spooler instead of a file
 0374         (* SHELL "spl -nj </pipe/SplIn"
 0394         (* OPEN #SplPipe,"/pipe/SplIn":write
 03BA         (* Now process data
 03CE         GET #NamPath,Record
 03D9         RUN PrintHeading(InPipe,"7/24/86")
 03ED         WRITE #InPipe,Record.Name
 03FA         WRITE #InPipe,Record.Add1
 0407         WRITE #InPipe,Record.Add2
 0414         WRITE #InPipe
 041A         WRITE #InPipe,Record.Greeting
 0427         WRITE #InPipe
 042D
 042E         (* Send some commands to the formatter
 0454         WRITE #InPipe,".fi"
 0460         WRITE #InPipe,Includer
 046A         WRITE #InPipe,".bp"
 0476         CLOSE #InPipe
 047C
 047D         (* Get output from formatter, send to spooler
 04AB         WHILE NOT(EOF(#OutPipe)) DO
 04B6           READ #OutPipe,Line
 04C0           WRITE #SplPipe,Line
 04CA         ENDWHILE
 04CE
 04CF 10      (* We return here after each EOF trap
 04F8         CLOSE #OutPipe
 04FE       NEXT Index
 0509       CLOSE #NamPath
 050F       CLOSE #SplPipe
 0515       END
 0517
 0518 100   (* EOF on pipe generates OS-9 Error #211
 0544       ErrNo=ERR
 054B       IF ErrNo<>211 THEN
 0558         PRINT Beep;
 055E         PRINT "System Error--->No. "+STR$(ErrNo)
 057B         END
 057D       ELSE
 0581         GOTO 10
 0585       ENDIF
 0587       END
```

Code Example 3: *Procedure to add front end to text formatter*

We Do Windows!

WARNING: This product may promote large incomes, high productivity, and excessive free time.

Magus, Inc. is proud to present Data&Windows: a window-oriented data-entry system based on a new, revolutionary design philosophy. The only problem is . . . what should we call it?

It is easy to learn and use, like a **panel generator**, because it allows you to draw your text, fields, and colors on the video display. But we can't call it a panel generator, because it supports full windowing and scrolling, and because screen and field information may be stored in your program files (.EXE) rather than separate data files.

It is flexible and powerful, like a **library-oriented programmer's toolkit**, but you are not restricted to "visualizing" your data-entry windows as you type page after page of code to set up borders, fields, text and highlighting. Our innovative approach (called **static windowing**) eliminates the need for replication of static data in dynamic memory.

It produces tight code, like a **YACC** (**Y**et **A**nother **C**ompiler **C**ompiler), but you don't have to tolerate a myriad of small program modules that need to be compiled and maintained. Instead, our "screen designer" creates Microsoft object files which you simply link with your applications.

Add to this new, superior design philosophy the fact that it has more features, produces tighter code, and yields higher performance than any of the above. Throw in a clear, concise user manual, a thorough on-disk tutorial, and some example programs. Top it off with a utility program that documents each screen and another that allows you to prototype (or simulate) your application before you write a single line of code. Now, what would *you* call it?

Let's settle on a single word.
Let's call it the "best."

But don't take our word for it. Order your demo disk today. You will receive a copy of the screen generator, the tutorial, and some documentation on the utility programs and library routines. Then make the decision yourself.

Or take advantage of our one-time introductory offer and get $100 discount if you order before March 31, 1987.

Call (713) 665-4109 for more information. Major credit cards accepted.

SCREEN DESIGNER:
Move/delete/center/fill/highlight block, global field redefinition, move/resize window and buffer, add/modify/move/delete field, insert/delete/undelete line, test/save/abort window, graphics characters, paint, jump-to-field, many cursor movement options, monitor switching, operating system calls, help. Set validation mode, highlight current field, scrolling by line or page, input mask, tabs, initial values. More. Up to 400 lines per screen.

FIELD DEFINITION:
Left-justify/right-justify/center, uppercase translation, built-in character validation, byte/integer/word/long/float/double/string/date field validation, retain data, auto-erase, protected fields, input required, use commas, use zeros/spaces, margin bell. User-defined character validations, pattern-matching validations, picture validations, and field types. More. Up to 9999 fields per screen.

LIBRARY ROUTINES:
Open, close, move, display, and refresh windows. Allow user to edit data fields in window, or to view and manipulate a window but not change data stored in it. Pull-down and pop-up menus. Read screen object file from disk. Intercept keyboard filter. Override default key actions. Automatic and manual refresh. Switch display device, erase all data fields on window, plot data onto fields or entire screens, retrieve data from fields or entire screens, screen image dump, retrieve and modify screen and field attributes, locate field, force use of bios. Direct interfacing with some bios interrupts, including cursor and mouse control. More. Mnemonic and simple to use.

REQUIREMENTS:
IBM PC/XT/AT/JR or true compatible, DOS 2.0 or later, at least 128K free RAM, and the Microsoft C, Pascal, or Fortran compiler or the IBM C compiler. Support is available for other C Compilers and the XENIX operating system. Call for specifics.

Circle **no. 336** on reader service card.

For More Information

(713) 665-4109

Data&Windows

Magus Inc.

Screen Designer

Let's Do Windows!

□ C □ Pascal □ FORTRAN

□ Please send __ copies @	$345.00
Introductory discount	**-100.00**
Your price	**245.00**
□ Data&Windows with Library Source Code	$695.00
Introductory discount	**-100.00**
Your price	**595.00**

Hurry! Introductory offer expires March 31, 1987.

Show Me More!
□ Send me a Demo $10.00

In Texas add 6.125% sales tax ________
Outside U.S. add 15.00 ________
Total enclosed $ ________

Enclosed is
□ Check □ Money Order
□ Visa □ MasterCard
Number ________
Expiration Date ________

Name ________
Company ________
Address ________
City ________
State ________ Zip Code ________

Send to:

MAGUS, INC.
4545 Bissonet Suite #114
Bellaire, TX 77401

Ctrace

The perfect companion for MIX C has arrived. MIX C makes it easy to write C programs. Now Ctrace makes it easy to get them working. Introducing Ctrace, the exciting new C source debugger with animated trace.

FUN AND EASY

Ctrace makes it so easy to debug your C programs that you'll love doing it. You no longer have to mess with assembly language or hex addresses. Ctrace presents your program in a form that's instantly familiar. Your C source code is displayed just as you wrote it. All your variables are displayed just as you named them. And wait till you see your program in action. Ctrace brings it to life on the screen. You'll see your variable values changing as you watch your source code executing. Ctrace shows you how your program works, or why it doesn't work. After one session with Ctrace, you'll wonder how you ever programmed in C without it.

UNIQUE ANIMATED TRACE

Ctrace has a unique animated trace feature that shows you the flow of execution in vivid detail. Not just line by line, but statement by statement. It's like watching the bouncing ball as the cursor moves over your C source code, highlighting each statement as it executes. Press the space bar to execute one statement at a time, or press the return key and watch it go. It's exciting and educational. Who says learning has to be boring?

SIMPLE OPERATION

Ctrace is easy to operate too. Commands are executed with a single keystroke. Help screens are available if you forget a command. Pop up menus list command options. You simply position the cursor to the desired option and press the return key. Pop up messages alert you when anything important happens. To use Ctrace, simply compile your program with the trace option turned on. The executable program file is created as normal. Ctrace doesn't affect the size or the behavior of the program. You can execute your program with or without the help of Ctrace.

4 VIEWS AT ONCE

Ctrace maintains 6 windows of information: source, output, variables, watch, symbols, and memory. You can view as many as 4 windows at the same time. The source window (top left) shows your C program. The output window (bottom left) shows the screen output from your program. The variable window (bottom right) shows all the variable names and values. The watch window (top right) shows the variables that you select along with any conditions you've defined. The symbols window shows the addresses of variables and functions. The memory window shows any area of memory using data types that you select. Eight different screen layouts are available at the touch of a key. You can even define your own screen layouts.

COMPLETE PROGRAM CONTROL

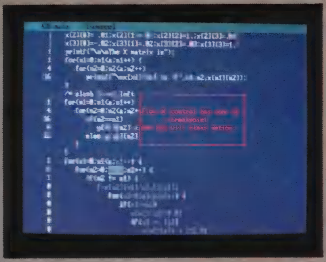
Ctrace gives you complete control of your program. Execution options are single step, trace speed, and full speed. You can insert breakpoints on an unlimited number of statements. Execution is temporarily halted when a break point is hit. You can then snoop around and see what your program has done to that point. You can even trace the flow of control backwards to see how your program got there. You can insert watch points on variable values. When the value of a variable satisfies the conditions you've defined, execution halts to let you examine your program. You can trace all functions or select just the ones you want to see.

THE RIGHT PRICE

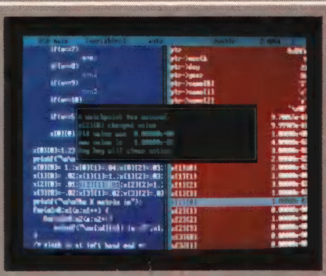
If you could buy a debugger like Ctrace anywhere else you would expect to spend major bucks. Fortunately nobody else has a debugger like Ctrace. It's only available from MIX Software. And that's great news because you know our prices are right. Ctrace is an incredible value at only $39.95. That's Right.

MIX software

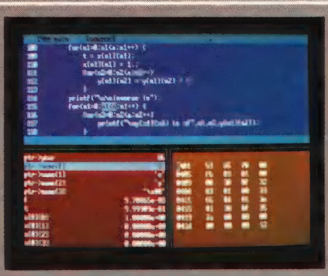
1132 Commerce Drive
Richardson, Tx. 75081
(214) 783-6001



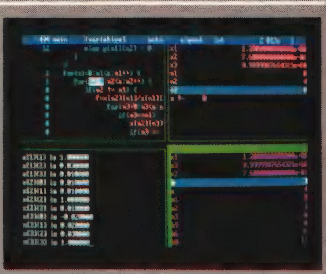
Source window with profile count showing number of times each statement has executed. Pop up message indicates break point has been hit.



Source and variables windows shown side by side. Pop up message indicates that a watch point condition has been satisfied.



Source, variables, and memory window. Memory window lets you view any area of memory using any data type.



Change colors to suit yourself. Ctrace works with monochrome, color, Hercules, and EGA cards. Works on IBM compatibles and any computer with an IBM compatible BIOS.

C for yourself

TO ORDER CALL TOLL FREE:
1-800-523-9520

For technical support and for orders inside Texas please call (214) 783-6001

Or Contact one of our Worldwide Distributors direct in:

Canada:	Saraguay	1-800-387-1288
France:	Info/Tech	1-43-44-06-48
Australia:	Techflow	047-586924
Switzerland:	DMB	CH-523-31817

System Requirements
Editor, C Compiler, & ASM Utility
MSDOS/PCDOS 2.0 or higher
128K Memory
1 Disk Drive
or CP/M 2.2 or higher (Z80)
55K Memory
1 Disk Drive (2 recommended)

System Requirements
Ctrace:
MSDOS/PCDOS 2.0 or higher
IBM compatible BIOS
256K Memory
1 Disk Drive

THE C COMPILER

You can see that Ctrace is not your typical debugger. It's easy to understand and simple to operate. Likewise, MIX C is not your typical C compiler. It's small and fast. In fact it's the only full feature C compiler that can be operated comfortably on floppy disks. And as you would expect, MIX C is easy to use. It produces a complete program listing with all errors clearly identified and explained.

Although it's small, MIX C is not a subset. MIX C supports the full K&R standard, including the extensions that are often omitted in other C compilers. MIX C comes complete with a comprehensive 460 page book, a library of more than 175 functions, a blazingly fast linker, and tools for optimizing your programs for minimal space or maximum speed. All of this is yours for the incredibly low price of $39.95. That's little more than the cost of most C books alone.

If you're just learning C, MIX C is your fastest, easiest, and cheapest way to master the language. If you've been frustrated by other C compilers, don't throw in the towel until you've tried ours. There's a world of difference. Our book includes a well written tutorial with lots of example programs. Our compiler includes the machine specific functions you need so you won't have to write them yourself. Compile and link operations take half as long with MIX C. That means you'll get your programs up and running twice as fast.

THE ASM UTILITY

Our ASM utility is available if you want to link assembly language functions to your C programs. It works with Microsoft's MASM or M80 assemblers. Macros make it easy! You can call assembly language functions just like C functions. You can even call C functions from assembly language. Lots of useful assembly language functions are included as examples. And the price is right at only $10.

THE SPLIT-SCREEN EDITOR

Another great companion to the MIX C compiler is our split-screen editor. It makes writing programs even faster and easier. With the MIX Editor, you can compile, link, and execute your program at the touch of a key. Compiling is fast because the MIX C compiler reads the program directly from memory. Correcting errors is easy because the editor automatically positions the cursor to the first error in the program.

The MIX Editor works just like Micropro's WordStar.® But through the magic of macros you can create your own custom version. You can map any key to any command. You can even define your own commands using the 100+ predefined commands. The split-screen feature is great for programming. You can edit two files at the same time and move text between files. It works great with any language. It has automatic line numbering for BASIC. It has auto indent for structured languages like Pascal and C. It even has fill and justify for English. All these features and more are yours for the incredibly low price of $29.95.

THE MIX C WORKS

The combination of Ctrace with MIX C makes C programming a real joy. MIX C provides the power of a compiler while Ctrace provides an execution environment that's more elegant than an interpreter. Add the ASM utility and our versatile split-screen editor to the package and you've got a terrific C programming system. We call it the MIX C Works. What's great is that you can buy all four products for a fraction of the cost of other C compilers alone. Yes, buy all four and we'll give you a big $29.95 discount off our already rock bottom prices. Only $89.90 for the MIX C Works. Now that's a deal. That's Right.

Product	Price	Total
_Ctrace	($39.95)	$ _____
_C Compiler	($39.95)	$ _____
_ASM Utility	($10.00)	$ _____
_Split-Screen Editor	($29.95)	$ _____
_The MIX C Works (includes all of above)	($89.90)	$ _____
Subtotal		$ _____
Texas Residents Add 6.125% Sales Tax		$ _____
Add Shipping Charges		$ _____

In USA: add $5 per order
In Canada: add $10 per order
Overseas: add $10 for editor
add $20 for compiler
add $30 for Works

Total of Your Order .... $ _____

30 Day Money Back Guarantee
Not Copy Protected

Please check method of payment

__Check __Money Order __MC/VISA

Card # _____ _____ _____ _____

Expiration Date _____________

Please give name of computer

____

Please check operating system

__ MSDOS/PCDOS __ CP/M

Please check disk size

__ 5¼" __ 3½" __ 8"

Please check disk format if CP/M

__ SSSD __ SSDD __ DSDD

Your Name _____________

Street _____________

City _____________

State _________ Zip _________

Telephone (_____) _____-_______

Country _____________

MIX software
1132 Commerce Drive
Richardson, Tx. 75081
(214) 783-6001

Ask about our volume discounts!
Dealer Inquiries Welcome

D

Macintosh Buttons and Amiga Gadgets

by Jan L. Harrington

When you put aside emotional reactions to the Macintosh and Amiga computers to take a more objective look at the two machines, it appears that, at least in concept, they are more alike than different. In addition to similarities in user interface standard, both use the 68000 microprocessor. Both have operating systems that are partially in ROM and partially on disk. Programmers who wish to adhere to the standard user interface make use of a set of system routines to create and manage windows, menus, graphics, and text.

The functions that the Macintosh and Amiga system routines provide are not equivalent, however. The Macintosh, for example, provides text handling routines not found on the Amiga. By the same token, the Amiga libraries contain animation routines not found on the Macintosh. The disparity between the functions provided by system routines presents challenges for programmers, especially if they are attempting to adapt software from one machine to the other.

This article explores the details of the standard Macintosh and Amiga user interfaces, examines the system routines programmers use to create those interfaces, and discusses those system routines through a pair of sample 68000 assembly-language programs that support a portion of the standard user interfaces.

Jan L. Harrington, 4002 Stearns Hill Rd., Waltham, MA 02154. Jan is an assistant professor in the Computer Science Department at Bentley College. She is the author of Macintosh Assembly Language: An Introduction.

The Mac's system routines support its user interface more completely.

User Interface Standards

The Macintosh standard user interface has caused many people to redefine what they mean when they say software is easy to use. You expect to be able to run a Macintosh program by double-clicking its icon on the Macintosh desktop. You expect to find at least Apple, File, and Edit menus present, and you expect those menus to behave in a consistent manner (they should "pull down" to expose the menu items, some of which may be associated with a keyboard equivalent). You expect to be able to use scroll bars to move throughout a document and to be able to simulate a click on an OK button by pressing the Return key. The Macintosh user interface standard is clearly defined in Chapter 2 of *Inside Macintosh*,[1] the extensive technical documentation for the machine.

The Amiga also supports a mouse-driven interface. Amiga applications that have icons can be run by double-clicking that icon from the Workbench window with the left mouse button. Amiga menus also pull down, the result of right mouse button action. Menu items can be associated with keyboard equivalents, and Amiga software has appeared with scroll bars.

The Amiga interface standard, however, is not as strictly defined as is the Macintosh standard. Suggestions for the interface do appear in the *Intuition*[2] manual, the part of the technical documentation that describes the routines supporting the windowing environment.

The window is central to both the Macintosh and Amiga. You can move windows about the screen (that is, within the same plane), move them relative to other windows on the screen (that is, from front to back), size them, and close them. All of these functions are initiated by mouse action on some portion of the window. Window movement within the plane is controlled by the drag region, that part of the window's title bar not taken up by the window title or other graphics. A click of the mouse button on a box at the far left of the title bar will close the window (the Macintosh calls it a "GoAway box," the Amiga a "CloseWindow gadget"). The Amiga window title bar also contains, at the far right, gadgets that move the window from front to back ("depth arrangement gadgets"); Macintosh windows move back one layer when another window is activated and brought to the front. In both machines, windows are sized with the overlapping boxes that appear in the lower-right corner of the window (the Macintosh calls it a "grow icon;" the Amiga calls it a "sizing gadget").

Both the Macintosh and Amiga collect information and give warnings using special windows. The Macintosh uses "dialog boxes" to collect information and "alerts" to give warnings. For example, a dialog box will appear to accept the name under which a file should be saved, and an alert box appears if you attempt to close a document window without

saving its contents. The Amiga uses "requesters" to collect information and "alerts" to let you know that something catastrophic has occurred. Requesters are analogous to dialog boxes, but whereas Macintosh alerts signal that a potential for harm occurs, Amiga alerts appear only after it's too late to recover. Amiga alerts are generally equivalent to Macintosh system alerts (generated by 68000 system errors). Usually you close requesters, dialog boxes, and Macintosh alerts by clicking on a small rectangle containing a message such as OK or Cancel. (The Amiga calls the small rectangles "gadgets"; the Macintosh calls them "buttons," which are a type of "control.")

Menus, too, are essential to both the Amiga and Macintosh user interfaces. Macintosh menu titles are visible at all times across the top of the screen in the "menu bar." The Amiga "menu strip," which also appears across the top of the screen, is visible only when you press the right mouse button.

Menus on both computers pull down; the menu choices appear in a box below the menu title as you drag the mouse pointer over them. Amiga menu items may also have submenus, which appear when the mouse pointer is dragged over the menu item. In either machine, menu items can be associated with keyboard equivalents—a single character that, when pressed in conjunction with a special modifier key (the command key on the Mac, the solid A key on the Amiga), simulates the selection of a menu item with the mouse.

Both computers maintain menu definitions as linked lists of data structures containing data needed to draw the menus. At any one time, the Macintosh supports only one menu list. You can make changes by inserting and removing menus from the list. The Amiga logically attaches menu lists to windows rather than to the screen, making it possible for many menu lists to be present in memory at any given time. Which menus appear when you depress the right mouse button therefore depends on which window is active at the time.

A Macintosh program that adheres to the standard user interface has at least three menus. The Apple menu appears at the far left of the menu bar; its title appears as an apple icon (an apple with a bite taken out of it). The Apple menu supports the Macintosh desk accessories. The second menu from the left is the File menu. The File menu opens, closes, saves, and prints files and exits from the program to the operating system. The third menu, the Edit menu, handles the text editing functions—Cut, Copy, Paste, Clear, and sometimes the Undo function as well.

The Amiga does not provide routines to implement its own standard interface recommendations.

Amiga programs that follow the interface suggestions made in the *Intuition* manual provide at least two menus in each menu strip. The leftmost menu is the Project menu, which is analogous to the Macintosh File menu—it manages opening, closing, saving, and printing of files as well as exiting from the program to the operating system. The second menu from the left is an Edit menu, which provides access to the same editing functions as the Mac's Edit menu.

Text editing standards are also an important part of both the Macintosh and Amiga user interfaces. Macintosh programs that support text entry of any kind, including even the entry of a single line into a dialog box, support cut, copy, and paste operations from a standard Edit menu. The editing functions are also present in programs that do no text editing because desk accessories use cut, copy, and paste even if an application does not. Amiga programs that require text entry (most notably word processors) support the same text editing functions as Macintosh programs do, but the absence of desk accessories means that Edit menus do not appear in programs that are not concerned with text.

Implementation

To explore the support the Macintosh and Amiga provide for their standard user interfaces, I wrote two programs in 68000 assembly language, one for each machine (see Listings One and Two, pages 64 and 69 for the Macintosh program and Listing Three, page 69, for the Amiga program). The programs are more or less equivalent in function. Each opens a window for text entry (the Amiga program also opens a custom screen) and creates menus (three for the Macintosh, two for the Amiga). Each supports the entry of text from the keyboard. The programs both terminate if you either select Quit from the appropriate menu or click the mouse pointer on the box that closes the window. Both also make extensive use of constants and data structure offsets from the include files supplied with the Macintosh 68000 Development System and the Amiga Macro Assembler, respectively.

The Macintosh system routines are invoked through the 68000's trap mechanism (all system calls are assembled to begin with *%1010*, which is then trapped by the microprocessor). The trap mechanism retrieves the actual location of the routine from a jump table, which is loaded from ROM to RAM at system start-up. Calls to system routines are performed with trap macros, all of which begin with an underbar (for example, *_GetRMenu*). The trap macros themselves are defined in the Macintosh's include files. Macintosh system routines are organized into "managers," which must be initialized before the routines are called. Lines 5–14 of Listing One initialize all the Macintosh managers.

Amiga system routines are contained in libraries. All libraries except the exec must be opened before they can be used. This includes the intuition library, which is opened at the top of Listing Three with a call to the system routine *OpenLibrary* (lines 44–51). Each library has a base address. Exec's base is fixed and assigned to the constant *_AbsExecBase*; all other library bases are returned by *OpenLibrary*. System routines are

(continued from page 41)

called as subroutines whose starting addresses are relative to the library base, which is placed in register *A6* before the call. Assuming that the correct address is in *A6*, calls to Amiga system routines are handled by the macro *callsys* (lines 4–6).

The functional differences between the two programs are the result of the system routines available on the two computers. The Macintosh program, for example, has a menu the Amiga program does not. The Apple menu is provided for the Macintosh's desk accessories. Implementation of the desk accessories is handled completely by a series of calls to system routines. The text in the Macintosh window will word wrap because the routines that manage the text editing environment automatically handle that function. The editing operations—cut, copy, and paste—have also been implemented because each is handled by a call to a single system routine.

The Amiga does not support desk accessories and so has no equivalent to the Macintosh's Apple menu. The Amiga also does not provide system routines for doing word wrap or the standard editing functions. Therefore, text entry in the Amiga window is exactly like using a typewriter. On the other hand, the Amiga window can be sized, moved about the screen, and moved back and forth in the plane. These functions are handled automatically by the Amiga's operating system; they do not need to be included in an application's code. Performing the same operations on the Macintosh requires including additional program code (calls to system routines).

The Amiga program is more than twice as long as the Macintosh program. There are two major reasons for this. First, the Macintosh can obtain parameters for creating data structures (for example, windows and menus) from a resource file. A resource file contains templates that describe the contents (for example, items to appear in a menu), location (for example, initial coordinates of a window), and characteristics (for example, menu items that should initially be disabled) of structures an application will use. The Macintosh program's resource file (Listing Two) contains templates for the three menus and one window. Second, although both the Macintosh and the Amiga maintain data about program objects in linked lists of data structures, Macintosh system routines perform most of the structure initialization and list management, whereas the Amiga leaves those functions to the programmer. The amount and type of programming required to achieve the same program function is therefore rather different on the two machines. You can see examples of these differences, especially in terms of creating menus and windows, when the two sample programs perform event trapping and do text I/O.

Creating Menus

The Macintosh can create a menu by calling the system routine *GetRMenu*. *GetRMenu* reads the template from the resource file, allocates space in RAM for the menu record, and loads the data into the appropriate locations in that menu record. Menu items are stored in a linked list. The routine requires one parameter (the resource file ID that identifies the menu) and space on the stack for a handle to the menu record. After the routine is called, an application pulls the handle from the top of the stack, where it has been placed by *GetRMenu*. Initializing a menu record therefore requires four lines of code. The template for the Macintosh program's File menu, for example, appears in lines 5–14 of Listing Two. The menu structures are actually initialized in lines 29–32 of Listing One.

Initializing a menu data structure for an Amiga program is considerably more complex. Assuming that the menu items are to be text rather than graphics, the text must first be loaded into intuition text structures. Each intuition text structure must be included in a menu item structure. The menu item structures are then assembled into a linked list whose head is incorporated into the menu data structure.

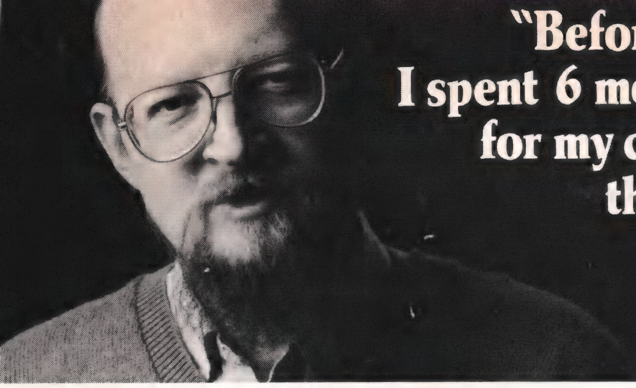
In the program in Listing Three, the intuition text structures are initialized in the subroutine *SetText* (lines 257–265), which is invoked by the macro *passtext* (lines 10–14). For each menu item, the text must be included in the program as a constant. The program must also allocate space for the intuition text structure (see lines 354–393 for the data structures associated with the Amiga program's menus). The intuition text structures for the Project menu, for example, are handled in lines 92–98.

Once the intuition text structures have been initialized, the menu items are initialized and chained into a linked list. The actual initialization is performed by the subroutine *SetItem* (lines 266–277), which is invoked by the macro *passitem* (lines 15–22). It is up to the programmer to allocate storage for each menu item data structure and to load the list pointers correctly. The Project menu's menu items are initialized and linked in lines 99–111.

The final step in the process is the initialization of the menu data structure. All the menus that will be present in a single menu strip are maintained in a linked list. Therefore, the initialization must include setting a pointer to the next menu in the list. The programmer must also determine coordinates for the physical position of the menu in the menu strip. The Amiga program sets up the Project menu in lines 112–123. After the Project menu has been completed, the entire process is repeated for the Edit menu.

The Macintosh requires the use of a separate system routine to insert menus into the linked list of menus that will appear in the menu bar at any given time. *InsertMenu* handles the insertion of a menu into the menu list. A programmer supplies either the handle of a menu after which this menu should be inserted or a parameter indicating that this menu should be last (that is, rightmost). Lines 33–35 of Listing One, for example, insert the File menu into the Macintosh's menu list.

Merely inserting a menu into the menu list does not, on either machine, display the menu. The Macintosh routine *DrawMenuBar* (see line 43 in Listing One) is equivalent in function to the Amiga routine *SetMenuStrip* (lines 149–152 in Listing Three). The Macintosh routine actually displays the menus currently in the menu list on the menu bar; the Amiga routine attaches the menu list to a window so



"Before I chose Microsoft C, I spent 6 months evaluating C compilers for my company. Now you can do the same in 2 hours."

Bill Davidsen
Software Engineer
Real Time and Operating Systems

"Call us. You can get Microsoft C or our comprehensive report on C by the day after tomorrow."

Bruce Lynch, President
The Programmer's Shop

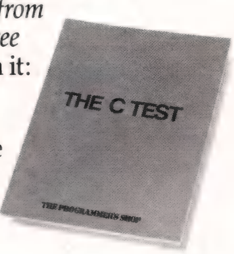
The security of thorough research. It took Bill Davidsen six months to thoroughly evaluate all C products before he selected Microsoft C. For him, its tight code and UNIX System V™ compatibility were exactly what he needed. And now Version 4.00 includes CodeView,™ a source-level windowing debugger.

Thanks to expert users like Bill, and The Programmer's Shop, you can enjoy that satisfied feeling of thorough product evaluation in just a few hours.

We recommend evaluating software by also getting detailed information from several different sources, including unbiased reports and reviews. Bill agrees completely.

In fact, he helped us compile the objective opinions of 4 magazines, 14 users and 3 industry analysts in a 16-page report on C: *The C Test.* It can help you be absolutely sure of making the choice that's best for you. And it's absolutely free.

C for yourself. As an objective evaluation by users and professionals alike, *The C Test* is one of the most comprehensive and informative reports currently available on C development tools. *It's only available from The Programmer's Shop. And it's yours free for the asking.* Here's what you'll find in it:



The C Test ▪ Detailed Tech Specs ▪ Benchmark Source Code ▪ Magazine Reviews ▪ Users' Feedback ▪ Performance Benchmarks ▪ User Study and Profiles ▪ Test Drive Survey Results ▪ 37 Compatible Products

And if you're looking for even more C support, Microsoft-compatible libraries for file management, graphics, screen control, object-oriented programming and other tools are ready to ship.

The best programs for less. We think the only way to serve you is to give you the best programming alternatives. The best recommendations for your needs. To deliver immediately. And this is how we do it.

We start by giving you a choice of over 62 programming language implementations and 174 support programs. All from the same source. All competitively priced.

Our informed programmers offer free advice whenever you call with any questions about any product.

And when you place an order, we can rush it to you in 48 hours or less. That's the kind of service and support our 10,000 customers have come to expect.

Because we've become a success by giving the best advice for free and selling the best software for less.

To order Microsoft C ($279) or for your free copy of *The C Test,* simply call the toll-free number below:

1-800-421-8006. In Massachusetts, call 1-800-442-8070.

MICROSOFT® C Compiler Version 4.00

MICROSOFT C COMPILER
- Produces fast executables and optimized code including elimination of common sub-expressions. NEW!
- Implements register variables.
- Small, Medium and Large Memory model libraries.
- Compact and HUGE memory model libraries. NEW!
- Can mix models with NEAR, FAR and the new HUGE pointers.
- Library routines implement most of UNIX System V C library.
- Start-up source code to help create ROMable code. NEW!
- Full proposed ANSI C library support (except clock). NEW!
- Link your C routines with Microsoft FORTRAN (version 3.3 or higher), Microsoft Pascal (version 3.3 or higher) or Microsoft Macro Assembler.
- Microsoft Windows support and MS-DOS 3.1 networking support.

MICROSOFT PROGRAM MAINTENANCE UTILITY. NEW!
- Rebuilds your applications after your source files have changed.
- Supports macro definitions and inference rules.

OTHER UTILITIES.
- Library Manager.
- Overlay Linker.
- EXE File Compression Utility.
- EXE File Header Utility.

MICROSOFT CodeView
WINDOW-ORIENTED SOURCE-LEVEL DEBUGGER. NEW!
- Watch the values of your local and global variables and expressions as you debug.
- Set conditional breakpoints on variables, expressions or memory; trace and single step.
- Watch CPU registers and flags as you execute.
- Debug using your original source code, the resulting disassembly or both intermingled.

Microsoft C comes with a 30-day money-back guarantee from The Programmer's Shop.

THE PROGRAMMER'S SHOP

The programmer's complete source for software, services and answers.

128 Rockland Street, Hanover, MA 02339 **(617) 826-7531**

Circle **no. 174** on reader service card.

MAC BUTTONS, AMIGA GADGETS
(continued from page 42)

that, when the window is active and the right mouse button is depressed, the menu strip appears.

There are both advantages and drawbacks to the Amiga's way of handling menus. The major drawback is the burden the Amiga places on the programmer. Establishing Amiga menus requires a great deal of work to initialize data structures and a great deal of care to ensure that pointers are stored properly. The Macintosh, on the other hand, isolates the programmer from dealing directly with the data structures and from list management. The trade-off is flexibility. Macintosh menu items are restricted to text (though a limited number of icons can appear with them), whereas Amiga menu items can be graphics (the intuition text structures can be replaced with graphics data structures). A Macintosh programmer has control only over the relative placement of a menu in the menu bar; an Amiga programmer can determine exactly where a menu should appear.

Creating Windows

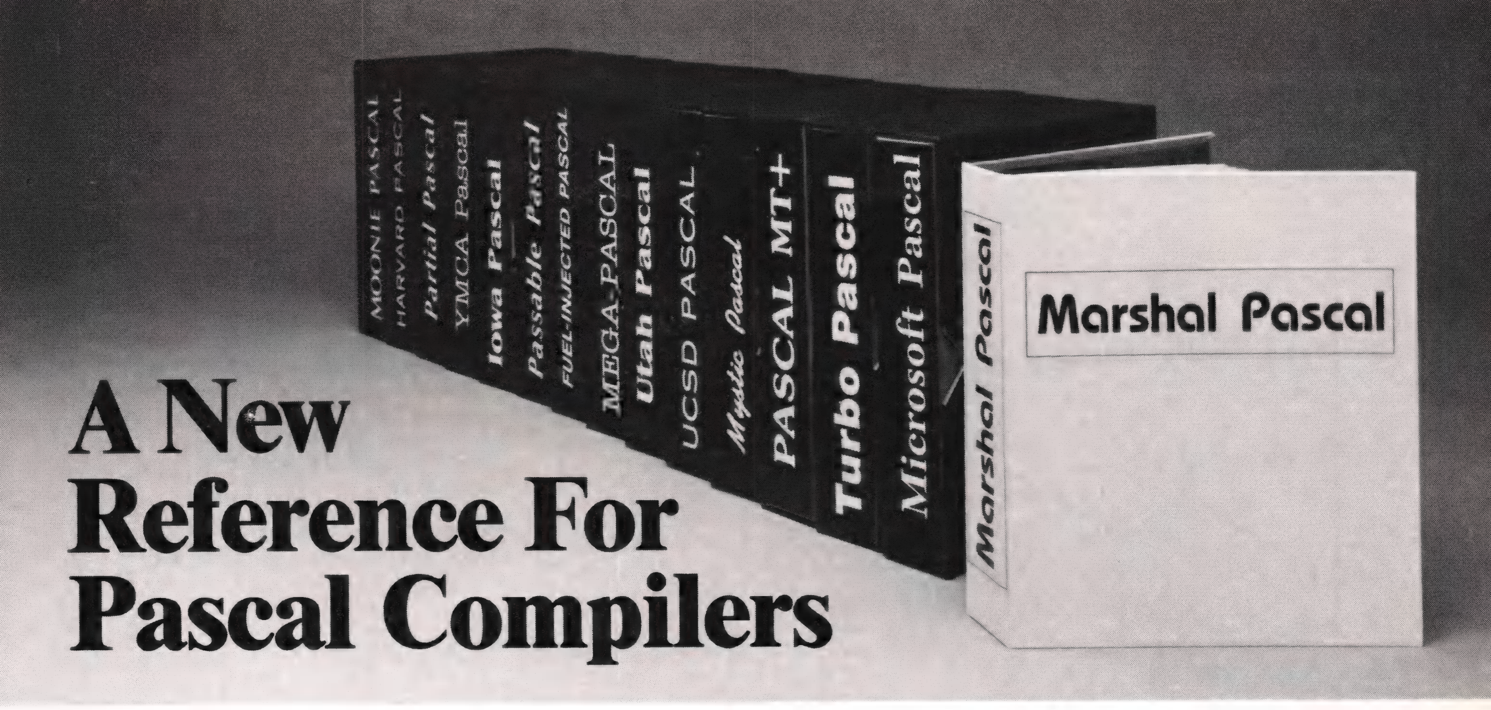
Macintosh windows can also be defined by templates in resource files (see lines 24–29 in Listing Two). An application can then create the window by pushing space for a window pointer on the stack, pushing three parameters, and calling *GetNewWindow* (see lines 44–49 in Listing One). The programmer must provide storage for the window record and for its pointer (lines 195–196).

Because the Amiga doesn't support resource files, an Amiga program must load a window data structure explicitly before calling the *OpenWindow* system routine, which actually creates the window (lines 69–91 in Listing Three). If the window is to appear in a custom screen rather than the default Workbench screen, the program must also first initialize a screen data structure and call *OpenScreen* (lines 52–68). Note that although it doesn't matter to the Macintosh whether windows or menus are created first, it does matter to the Amiga. Amiga menus are attached to windows, not to the screen, and therefore a menu strip is useless unless a window has previously been created to which it can be attached.

Event Trapping

Event trapping on the Macintosh and Amiga is similar in principle but somewhat different in detail. The general idea is to somehow let the computer know which events are of importance and then to enter a wait state until a desired event occurs. Once an event has been recorded, a program must identify which type of event has been posted and take action based on that particular event.

The Macintosh posts events to a system event queue. Events of interest to an application program (that is, those that aren't handled automatically by the system) are passed on to the program. An application program checks its queue repeatedly



A New Reference For Pascal Compilers

	Ackerman		Sieve		I/O		Gauss-Seidel		Floating Point	
Marshal Pascal	11.9	5.1K	4.8	3.4K	1.9	6.8K	5.7	11.5K	10.5	5.3K
IBM Pascal	12.4	34.7K	11.7	27K	2.6	28K	7.6	34.6K	10.5	21K
Turbo Pascal	22.7	11.6K	14.2	11.5K	2.2	12.5K	4.7	13.5K	28	11.4K
Microsoft C 4.0	15.9	9.3K	5.8	6.5K	1.9	8.9K	6.0	23.6K	33	19.6K
	Sec.	Code Size	Sec.	Code Size	Min.	Code Size	Sec.	Code Size	Sec.	Code Size

Unparalleled Speed and Power

Marshal Pascal™ is the most highly code-optimized Pascal compiler for PCs. Period. In fact, Marshal Pascal produces code so fast and compact that even the most efficient C compilers fall behind in performing many operations. Also, Marshall Pascal is an ISO implementation thus offering portability to other computer environments. You may address as much memory as your operating system allows and a variety of memory models are supported. Among the useful extensions included are: separate compilation of modules (both Modula-2 and Pascal forms), structured constants and structured function values, variable-length string types and procedural parameters.

Turbo Pascal® Translator

Our translator brings your Turbo Pascal software over to an ISO/Marshal-readable format. You can watch your present Turbo programs run in a fraction of their former time and code space!

8087-80287 Support

Marshall Pascal supports the Intel 8087-80287® math processors *inline*. If you don't have the math chip, then '87-287 code emulation is a provided option.

Marshal Language Systems

1136 Saranap Ave., Ste. P, POB 2010, Walnut Creek, CA 94595

Microsoft Linkability

Marshal Pascal's true relocatable linker allows you access to the Microsoft family of languages and assemblers. A flexible object code librarian and powerful overlay capabilities are also included.

Powerful Compile Options

Marshal Pascal gives you a number of compile options, including an optimization by-pass for speedier compiles, I/O "fine-tuning", constant folds and a syntax evaluator just to name a few. A wealth of compile-time checks permits you to find the more subtle logic errors, reducing debugging time enormously.

Space does not permit us to list all the features of Marshal Pascal, all of which are provided USER options, not additional COST options. Marshal Pascal is not broken apart to make many products out of what should be only one good one.

The Price?

$189. *includes everything.*

Supports PC-DOS®, MS-DOS®, CP/M-86® and Concurrent DOS®.

To order your copy of **Marshal Pascal,** call: **(800) 826-2222**
In California: **(415) 947-1000**

Circle **no. 317** on reader service card.

SAS Institute Inc. Announces

Lattice C Compilers for Your IBM Mainframe

Two years ago...

SAS Institute launched an effort to develop a subset of the SAS® Software System for the IBM Personal Computer. After careful study, we agreed that C was the programming language of choice. And that the Lattice® C compiler offered the quality, speed, and efficiency we needed.

One year ago...

Development had progressed so well that we expanded our efforts to include the entire SAS System on a PC, written in C. And to insure that the language, syntax, and commands would be identical across all operating systems, we decided that all future versions of the SAS System—regardless of hardware—would be derived from the same source code written in C. That meant that we needed a C compiler for IBM 370 mainframes. And it had to be good, since all our software products would depend on it.

So we approached Lattice, Inc. and asked if we could implement a version of the Lattice C compiler for IBM mainframes. With Lattice, Inc.'s agreement, development began and progressed rapidly.

Today...

Our efforts are complete—we have a first-rate IBM 370 C compiler. And we are pleased to offer this development tool to you. Now you can write in a single language that is source code compatible with your IBM mainframe and your IBM PC. We have faithfully implemented not only the language, but also the supporting library and environment.

Features of the Lattice C compiler for the 370 include:

- **Generation of reentrant object code.** Reentrancy allows many users to share the same code. Reentrancy is not an easy feature to achieve on the 370, especially if you use non-constant external variables, but we did it.
- **Optimization of the generated code.** We know the 370 instruction set and the various 370 operating environments. We have over 100 staff years of assembler language systems experience on our development team.
- **Generated code executable in both 24-bit and 31-bit addressing modes.** You can run compiled programs above the 16 megabyte line in MVS/XA.
- **Generated code identical for OS and CMS operating systems.** You can move modules between MVS and CMS without even recompiling.
- **Complete libraries.** We have implemented all the library routines described by Kernighan and Ritchie (the informal C standard), and all the library routines supported by Lattice (except operating system dependent routines), plus extensions for dealing with 370 operating environments directly. Especially significant is our byte-addressable Unix®-style I/O access method.
- **Built-in functions.** Many of the traditional string handling functions are available as built-in functions, generating in-line machine code rather than function calls. Your call to move a string can result in just one MVC instruction rather than a function call and a loop.

In addition to mainframe software development, you can also use our new cross-compiler to develop PC software on your IBM mainframe. With our cross-compiler, you can compile Lattice C programs on your mainframe and generate object code ready to download to your PC.

With the cross-compiler, we also offer PLINK86™ and PLIB86™ by Phoenix Software Associates Ltd. The Phoenix link-editor and library management facility can bind several compiled programs on the mainframe and download immediately executable modules to your PC.

Tomorrow...

We believe that the C language offers the SAS System the path to true portability and maintainability. And we believe that other companies will make similar strategic decisions about C. Already, C is taught in most college computer science curriculums, and is replacing older languages in many. And almost every computer introduced to the market now has a C compiler.

C, the language of choice...

C supports structured programming with superior control features for conditionals, iteration, and case selection. C is good for data structures, with its elegant implementation of structures and pointers. C is conducive to portable coding. It is simple to adjust for the size differences of data elements on different machines.

Continuous support...

At SAS Institute, we support all our products. You license them annually; we support them continuously. You get updates at no additional charge. We have a continuing commitment to make our compiler better and better. We have the ultimate incentive—all our software products depend on it.

For more information...

Complete and mail the coupon today. Because we've got the development tool for your tomorrow.

SAS®

SAS Institute Inc.
SAS Circle, Box 8000
Cary, NC 27511-8000
Telephone (919) 467-8000 x 7000

I want to learn more about:

☐ the C compiler for MVS software developers
☐ the C compiler for CMS software developers
☐ the cross-compiler with PLINK86 and PLIB86

today...so I'll be ready for tomorrow.

Please complete or attach your business card.

Name ______________________

Title ______________________

Company ______________________

Address ______________________

City ______________ State ________ ZIP ________

Telephone ______________________

Mail to: SAS Institute Inc., Attn: CC, SAS Circle, Box 8000, Cary, NC, USA. 27511-8000. Telephone (919) 467-8000, x 7000

DDJ 1/87

MAC BUTTONS, AMIGA GADGETS

(continued from page 44)

with the routine *GetNextEvent* (lines 62–65 in Listing One) to determine if an event has been posted. If *GetNextEvent* returns a Boolean result of *FALSE*, then the program simply branches to the top of the event loop (line 59) to check again. Assuming that an event has been posted, information about the event is stored in an event record. The program can then use information from that record to identify the exact type of event (see lines 69–74). In some cases, events not of interest to the application can appear in the event queue. When that occurs, the program simply ignores the event (line 75). When a desired event is identified, however, Macintosh programs generally branch to submodules, each of which processes a single event type. When the event has been handled, the program returns to the event loop to idle until another event is posted to the event queue.

The Amiga reports events via message ports, which must be initialized before they can be used. The *OpenWindow* routine creates an intuition message port, but ports must be created explicitly for the console device, which will be used for text I/O. The subroutine *CreatePort* (lines 278–307 in Listing Three) allocates a signal bit for a new port, allocates memory for the port's data structure, initializes a task control block for the port, and adds the port to the linked list of current message ports.

Amiga programs do not need a program loop to idle while waiting for an event. Instead, they can use the system routine *Wait*, which idles until a desired event occurs. *Wait* must be supplied with the signal bits assigned to each of the input ports that should be monitored for events. In the sample Amiga program, that includes the signal bit for the intuition message port and the signal bit for the console read port (see lines 184–193 in Listing Three).

Unlike the Macintosh, the Amiga doesn't report all types of events automatically. In line 77, the system is instructed to report only two of the types of events that may occur when this window is active—a click in the window close gadget and a selection from a menu. Therefore, any event detected by *Wait* should be an event useful to the program.

If a Macintosh program identifies a selection from a menu (a "mouse-down" event in the menu bar), the program is faced with the problem of identifying which item from which menu has been selected. A single system routine, *MenuSelect* (lines 125–127 in Listing One), returns both the menu's resource ID and the number of the menu item. *MenuSelect* uses a field from the event record as input—the coordinates of the mouse pointer when the selection was made. The menu number, returned in the high-order word of the long-word result, is then isolated from the menu item, which is returned in the low-order word (lines 128–130). Finally, the menu number can be used

(continued from page 47)

to identify the precise menu posting the event.

An Amiga program must also identify which menu and which item have been selected. The Amiga, though, reports only that an intuition event has occurred. The program must retrieve the input message with the system routine *GetMsg* (lines 205–208 in Listing Three) to determine exactly which of the intuition events requested in the window data structure has been detected. Once the type of event has been recovered (line 211), it can be compared against the desired types of events (lines 212–215) in the same way as the Macintosh program identifies events. The menu number and menu item (and the menu subitem, if applicable) are stored in a 16-bit field in the message data structure. The menu number is in bits 0–4 and can be isolated by masking off the 11 high-order bits (line 222). The menu item is in bits 5–10 (see lines 225–226).

The sample Macintosh program supports activities from all three menus. The Apple menu's desk accessories are handled by system routines. *GetItem* (lines 140–143 in Listing One) retrieves the name of the desk accessory that has been selected. *OpenDeskAcc* (lines 144–147) actually opens the program. Once the desk accessory has been opened, subsequent events in that program are posted to the event queue and detected as mouse-button down events in a system window (lines 104–105). In that case, the system routine *SystemClick* (lines 111–113) processes the event without further intervention from the application program. For the purposes of the sample program, only the Quit item is actually trapped from the File menu (lines 176–177); all other options simply return to the event loop. The *CloseAndQuit* routine (lines 187–191) frees the space used for text storage (*TEDispose* in lines 187–188), closes the window (*CloseWindow* in lines 189–190), and then returns to the Finder. The Edit menu, which is fully implemented with system routines, is discussed later in this article in the context of text editing.

The Amiga program also handles only the Quit item from its Project menu. Its *CloseAndQuit* routine (lines 229–240 in Listing Three) first removes any unprocessed console events with the system macro *ABORTIO* (this macro is found in the Amiga include files), then closes the console device (*CloseDevice* in lines 231–233), closes the window (*CloseWindow* in lines 234–236), and finally closes the custom screen (*CloseScreen* in lines 237–239) before returning to the operating system. The Edit menu is not implemented because the Amiga does not have system routines for text editing.

Text Editing

The environment for text editing is very, very different on the Macintosh and Amiga. Other writers have glossed over the Macintosh's text editing abilities (for example, see the September 1986 issue of *Byte* magazine[3]), but it is in this area that the difference between the Mac and Amiga system routines is glaringly apparent. The Macintosh's system routines for text editing are simple and elegant. The Amiga has nothing comparable; Amiga text I/O relies on low-level device communication.

To do text I/O, a Macintosh program allocates a text edit record with the system routine *TENew* (see lines 52–56 in Listing One). The text edit record is associated with a window, though the text stored in the text edit record can be much larger than what can be seen in the window at any given time. A call to *TEActivate* (lines 57–58) makes the text edit window active and draws the straight-line cursor as the text entry point. Repeated calls to *TEIdle* (line 61), usually within the program's event loop, ensure that the cursor blinks regularly.

When a text edit window is active, Macintosh programs generally assume that key-down events not associated with the command key represent text to be both displayed on the screen and stored in the text edit record. Whenever such an event is detected, the key pressed is stored in the event record. *TEKey* (lines 79–81 in Listing One) displays that character at the current cursor position, adjusting word wrap as necessary, and

inserts the character into the text edit record in RAM.

Mouse-down events in active text edit windows signal that the user wishes to either change the position of the straight-line cursor or identify a block of text for editing activities. The Macintosh refers to blocks of code or the straight-line cursor as the "selection range." Setting the selection range requires calls to two system routines. *GlobalToLocal* (lines 115–116) is a graphics routine that takes the point where the mouse-down event occurred, which is expressed in global screen coordinates, and translates it to the text edit window's local coordinate system. The transformed coordinates can then be passed to *TEClick* (lines 117–123), which actually sets the selection range.

The text editing functions listed in the Macintosh's standard Edit menu are each available as a single system routine that bases its operation on the current selection range. *TECut* (lines 156–159) deletes the current selection range from the screen, adjusting word wrap and the text edit record. The deleted text is stored in RAM in an area known as the "Clipboard." Cut, and any other operations that write to the Clipboard, erase the previous contents of the Clipboard. *TECopy* (lines 161–165) writes the current selection range to the Clipboard but does not affect the screen or text edit record. *TEPaste* (lines 166–169) inserts whatever it finds on the Clipboard at the current selection point (either at the straight-line cursor or after the selection range). Both the screen display and text edit record are adjusted. *TEClear* (lines 171–175) deletes the current selection range without affecting the Clipboard. These functions also provide "intelligent cut and paste," automatically adjusting spacing between words when text is either deleted, cut, or pasted.

There are two major ways to do text I/O on the Amiga—either via the console device, which processes keystrokes before passing them on to the system, or via the RAW device, which transmits unprocessed key codes. For simple text I/O, the console device is easier to use because it automatically manages special keys such as the backspace. To use the console device, an Amiga program must, as discussed earlier, allocate two message ports—one for input and one for output. These message ports are then incorporated into standard I/O request blocks, the data structures that are actually used for I/O. The subroutine *CreateStdIO* (lines 308–319, Listing Three) allocates memory for a standard I/O data structure and initializes the data structure with the address of its message port. Finally, the console device can be opened by the subroutine *OpenConsole* (lines 320–330). Note that the call to the system routine *OpenDevice* (line 327) returns a pointer to the device data structure in a field of one of the standard I/O request blocks. In other words, the device is linked to the I/O request blocks rather than the I/O request blocks being linked to the device. If the console is opened successfully, a solid block cursor appears in the upper left-hand corner of the window to which the console is attached. The cursor does not blink.

The system routine *SendIO*, which appears in the subroutine *QueueRead* (lines 332–337), issues a request for console input. *SendIO* uses data from the input standard I/O request block, including the type of operation to perform (line 332), the place where input should be stored (line 333), and the number of bytes to input (line 334).

The subroutine *ConPutChar* (lines 338–343) handles console output. Using the system routine *DoIO* (line 342), it displays a single character at the current cursor position and moves the cursor to the right. If the cursor is pushed past the right edge of the window, it drops to the leftmost position on the line below. *ConPutChar* does not do word wrap, nor does it store the text displayed in main memory. Subsequent calls to *QueueRead* reuse the same storage space for input characters. As mentioned before, the Amiga also has no system routines for text editing. Therefore, although the backspace key can erase whatever character is displayed to the left of the cursor, no other editing is possible. To implement the standard text editing functions, programmers must write their own routines to do word wrap, manipulate the selection range, store text in main memory (or on disk if applicable), manage a clipboard, and do the editing operations.

The Bottom Line

It might be unfair to base an evaluation of the system routines of the Macintosh and Amiga simply on the subset of the routines designed to manipulate the user interface. On the other hand, the programming strategies used to implement the standard user interfaces are similar to those required for other system operations. In general, the Macintosh routines isolate programmers from low-level tasks such as list manipulation and initialization of data structures (File Manager parameter blocks are notable exceptions). Although this reduces the burden placed on the programmer, it can decrease the programmer's flexibility. The Macintosh routines are also more complete in terms of their support for the documented user interface. The effect is again to reduce the burden placed on the programmer.

On the Amiga, the intuition library provides routines for the standard user interface. Although support for screen, windows, menus, and fonts is available, there is a great gap in terms of text editing. In other words, the Amiga does not provide system routines to fully implement its own standard interface recommendations. As someone who writes more programs that rely on text manipulation than on graphics, I believe that this is a serious deficiency. It is true that the Amiga performs some functions "automatically" for which a Macintosh program must include code (for example, moving and sizing windows). Nonetheless, the Macintosh does include system routines to handle those functions.

Notes

1. Caroline Rose et al., *Inside Macintosh* (Reading, Mass.: Addison-Wesley, 1985).

2. Robert J. Mical and Susan Deyl, *Intuition: The Amiga User Interface* (Commodore-Amiga Inc., 1985).

3. Adam Brooks Webber, "Amiga vs. Macintosh," *Byte* (September 1986): 249–257.

DDJ

(Listings begin on page 64.)

THE PROGRAMMER'S SHOP

provides complete information, advice, guarantees and every product for Microcomputer Programming.

COBOL SPECIAL

Call our COBOL Specialist to discuss compilers & support products to see what fits your background & interests.

Order before 1/31/87 and mention this ad for these special prices.

By MicroFocus	List	Normal	SPECIAL
Level II	1500	1350	589
Professional	3500	2695	1795
with Sourcewriter —			
Professional	5000	4554	2499
VS Workbench	6000	5349	4289
Microsoft COBOL	700	499	459
Ryan McFarland COBOL	950	699	595
COBOL-8X	1250	995	785

C Support-Systems

Basic-C Library by C Source	PC	$139
C Sharp - well supported. Source, **realtime,** tasks, state system	PC	$600
C ToolSet - DIFF, xref, source	MS	$ 95
The HAMMER by OES Systems	PC	$179
Lattice Text Utilities	MS	$ 95
Multi-C - multitasking	PC	$149
PC LINT-Checker. Amiga $89	MS	$107
SECURITY LIB-add encrypt to MS C, C86 programs. Source $229	PC	$115
Quickshell - script compiler	PC	$349

C-Screens. Windows. Graphics

C Power Windows by Entelekon	PC	$119
dBASE Graphics for C	PC	$ 69
Curses by Lattice	PC	$ 99
ESSENTIAL GRAPHICS - fast,	PC	$209
GraphiC - mono version	PC	$217
GraphiC - new color version	PC	$295
Greenleaf Data Window	PC	$199
w/source	PC	$369
Multi-Windows - use w/ Multi-C	PC	$295
Vitamin C - screen I/O	PC	$129
Windows for C - fast	PC	$159
Windows for Data - validation	PC	$239
ZView - screen generator	MS	$189

Debuggers

Advanced Trace-86 by Morgan Modify ASM code on fly.	PC	$129
CODESMITH - visual, modify and rewrite Assembler	PC	$107
C SPRITE - data structures	PC	$139
DSD87 - by Soft Advances	PC	$ 89
Periscope I - own 16K	PC	$249
Periscope II - symbolic, "Reset Box," 2 Screen	PC	$119
Pfix-86 Plus Symbolic Debugger by Phoenix - windows	PC	$249
Showcase by Test Software	PC	$135
Software Source by Atron - Lattice, MS C, Pascal, Windows single step, 2 screen, log file.	MS	$115
w/Breakswitch		$199

FEATURE

Pascal-2 - Perhaps tightest compiler for MSDOS. Mainframe background & power. MS compatible. Complete environment. Turbo translator. Get detailed specs. PC $ 329

Fortran & Supporting

ACS Time Series	MS	$429
Forlib + by Alpha - graphics and file routines, comm.	MS	$ 59
MACFortran by Microsoft	MAC	$229
MS Fortran link to C	MS	$209
No Limit - Fortran Scientific	PC	$119
RM/Fortran - enhanced "IBM Professional Fortran"	MS	$389
Scientific Subroutines - Matrix	MS	$149
Statistician by Alpha	MS	$259
Strings and Things - register, shell	PC	$ 59

Multilanguage Support

BTRIEVE ISAM	MS	$199
BTRIEVE/N-multiuser	MS	$469
CODESIFTER - Profiler.	MS	$ 99
HALO Graphics - 115 + devices. Animation, engineering, business. Any MS language, Lattice, C86	PC	$217
Informix - by RDS	PC	$679
Informix 4GL - application builder	PC	$849
PANEL - Create screen with editor, generates code. Full data validation, no royalties. Xenix $539,	MS	$219
PolyLibrarian by Polytron	MS	$ 85
PVCS Version Control	MS	$329
QMake by Quilt Co.	MS	$ 89
Rtrieve - Xtrieve option	MS	$119
Screen Sculptor - slick, thorough, fast, BASIC, PASCAL.	PC	$ 99
Xtrieve - organize database	MS	$199
ZAP Communications - VT 100, TEK 4010 emulation, file xfer.	PC	$ 89

Pascal and Supporting

ALICE - learn Pascal, Turbo compatible, interpreter	PC	$ 79
Exec - Chain Programs	MS	$ 85
MetaWINDOW - graphics toolkit	PC	$135
Microsoft PASCAL - faster	MS	$199
MICROTEC PASCAL - 5 memory models, "Iterators", 65 bit 8087 strings	MS	$665
Pascal Pac with Tidy - formatter, utilities	PC	$ 69
Pascal Tools - strings, screen	PC	$109
Pascal Tools 2 - by Blaise	MS	$ 85
Pfas - Portable Isam	MS	$185
TurboHALO - 150 routines, IBM EGA, Hercules, more	PC	$ 85
USCD Pascal - native code	MS	$ 69

RECENT DISCOVERY

TransLISP PLUS - with C INTERFACE, 400 + COMMON LISP functions.
Optional UNLIMITED Runtime $ 150
PLUS for MSDOS $ 179

Other Languages

APL*PLUS/PC	PC	$ 469
Artek ADA Compiler - DOD standard minus multitasking	PC	$ 469
CLIPPER-dBASE Compiler	MS	$ 429
Correct FORTH - ROMable	PC	$ 80
ED/ASM - 86 by Oliver	PC	$ 85
Lattice RPG II Compiler	PC	$ 719
MacASM - fast	MAC	$ 99
MasterForth - Forth '83	MAC or PC	$ 109
Microsoft MASM - faster	MS	$ 98
Modula 2 by Volition Systems	MS	$ 250
Modula-2/86 Compiler by Logitech w/ 8087 ($ 99), 512K ($145)	PC	$ 62
Pasm - by Phoenix	MS	$ 149
SNOBOL4 + - great for strings	MS	$ 85
Turbo Edit/ASM - by Speedware	PC	$ 85

Xenix-86 & Supporting

Basic - by Microsoft	$ 249
Cobol - by Microsoft	$ 659
Fortran or Pascal - by Microsoft	$ 489
MicroFocus Lev. II Compact COBOL	$ 795
Xenix Complete Development System	$1049

Other Products

386 Assembler/Linker	PC	$ 459
ASMLIB - 170 + routines	PC	$ 135
BSW Make - like UNIX make	MS	$ 85
Dan Bricklin's Demo Program	PC	$ 65
dBrief - Customize BRIEF for dBASE development. with BRIEF $275.	PC	$ 95
H Test/H Format - XT Fix	PC	$ 89
Interactive Easyflow-HavenTree	PC	$ 129
Link & Locate - tools to work with Intel and Tektronix projects.	MS	$ 329
LMK - like UNIX make	MS	$ 149
Microsoft Windows	PC	$ 75
Microsoft Windows Software Development Kit	PC	$ 349
MKS Toolkit - Unix, vi, awk	PC	$ 119
Opt Tech Sort - sort, merge	MS	$ 119
PMaker - by Phoenix	PC	$ 99
Polymake by Polytron	MS	$ 79
PS MAKE by UniPress	MS	$ 79
SECRET DISK by Lattice	PC	$ 95
SET:SCIL - manager revisions	PC	$ 299
Shrink/Shrinkem - put more files on disk with spacemaker	PC	$ 150
SoftEst - Manage projects.	MS	$ 350
Synergy-Create user interfaces	MS	$ 375
Texsys - control source	MS	$ 89
Visible Computer: 8088 - Simulates demos or any .exe. com, Debugger. 350 pg. tutorial, unprotected	PC	$ 69

Note: All prices subject to change without notice.
Mention this ad. Some prices are specials. Ask about COD and POs. Formats: 3" laptop now available, plus 200 others. UPS surface shipping add $3/item.

Call for a catalog, literature, advice and service you can trust

HOURS
8:30 AM - 8:00 PM EST.

800-421-8006

THE PROGRAMMER'S SHOP™
128-D Rockland Street, Hanover, MA 02339
Mass: 800-442-8070 or 617-826-7531 11/86

"I like your straightforward, open evaluations, comments and selection."

Chris Chapman
Practical Solutions Software

Listing One *(Text begins on page 22.)*

```
; Listing One
; FLINT's system macros

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                       ;
;  MACRO definition file for FLINT as implemented on the               ;
;                       SAGE II microcomputer.                          ;
;                                                                       ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

        .MACRO      XSTOP       ;   determines the environment (if any)
                                ;   into which the exit is made
        .WORD       4EF9H       ;
        .WORD       00FEH       ;   exit to PROM monitor/debugger
        .WORD       0030H       ;
        .ENDM

        .MACRO      TGET        ;   gets a character from the terminal
                                ;   and puts its ASCII value in the
                                ;   lower byte of D0
        .WORD       4EB9H       ;
        .WORD       00FEH       ;   PROM BIOS call
        .WORD       0008H       ;
        .ENDM

        .MACRO      TPUT        ;   sends the character whose ASCII value
                                ;   is in the lower byte of D0 to the
                                ;   terminal
        .WORD       4EB9H       ;
        .WORD       00FEH       ;   PROM BIOS call
        .WORD       0014H       ;
        .ENDM

        .MACRO      GETBLOCK            ;   loads 1024 bytes from the disk block
                                        ;   numbered by the value in CBLOCK into
                                        ;   a buffer whose address is DISKBUF
        MOVE.L      A4,-(A7)            ;   save A4
        MOVE.W      CBLOCK,-(A7)        ;   push block #
        LEA         DISKBUF,A0          ;
        MOVE.L      A0,-(A7)            ;   push buffer address
        MOVE.L      #1024,-(A7)         ;   push length of transfer (1024 bytes)
        MOVE.W      #1,-(A7)            ;   push drive #
        .WORD       4EB9H               ;
        .WORD       00FEH               ;   PROM BIOS call for diskread
        .WORD       0028H               ;
        MOVEA.L     (A7)+,A4            ;   recover A4
        .ENDM

        .MACRO      SAVBLOCK            ;   saves the contents of a 1024 byte
                                        ;   buffer whose address is DISKBUF onto
                                        ;   the disk block numbered by the value
                                        ;   in CBLOCK
        MOVE.L      A4,-(A7)            ;   save A4
        MOVE.W      CBLOCK,-(A7)        ;   push block #
        LEA         DISKBUF,A0          ;
        MOVE.L      A0,-(A7)            ;   push buffer address
        MOVE.L      #1024,-(A7)         ;   push length of transfer (1024 bytes)
        MOVE.W      #1,-(A7)            ;   push drive #
        .WORD       4EB9H               ;
        .WORD       00FEH               ;   PROM BIOS call for diskwrite
        .WORD       002CH               ;
        MOVEA.L     (A7)+,A4            ;   recover A4
        .ENDM
```

End Listing One

Listing Two

```
; Listing Two
; The FLINT Interpreter

        .NOLIST
        .NOSYMTABLE
        .NOMACROLIST
        .NOPATCHLIST
        .INCLUDE  MACROS.TEXT
        .LIST

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;                                                                                  ;
; FLINT     (Forth-Like INterpreter and Threader)                                  ;
;                                                                                  ;
;  We note the effect that the execution of a word has on the stack by a           ;
;  conventional shorthand. Before and after lists of the relevant stack            ;
;  parameters are given in "rightmost is topmost" order separated by               ;
;  "-->". An address is denoted by "a", an integer by "n", and a flag by           ;
;  "fl". Parameters in parentheses may or may not be present.                      ;
;                                                                                  ;
;    word              action                                  stack effects       ;
;                                                                                  ;
;   TOKEN  gets token from input buffer                                            ;
;                                                               _                  ;
;   SEARCH            searches dictionary for current token     a --> (a),fl       ;
;                     pushes its code address and true flag                        ;
;                     or else false flag                                           ;
;                                                                                  ;
;   EXECUTE           executes routine whose code address       a --> __           ;
;                     is on stack                                                  ;
;                                                                                  ;
;   NUMBER            determines if a token whose address is                       ;
;                     popped from stack represents a number     a --> (n),fl       ;
;                     if so pushes value and true flag or else                     ;
;                     pushes false flag                                            ;
```

```
;                                                                              ;
;    WHAZZAT           sends ? to terminal                           _         ;
;                                                                              ;
;    HEADER            makes a dictionary header for the next                  ;
;                      token in the input stream (the name of        _         ;
;                      a word being defined).                                  ;
;                                                                              ;
;    :                 calls HEADER and then sets the compile                  ;
;                      mode (colon) flag                                       ;
;                                                                              ;
;    COMPILE           writes code for JSR in the dictionary                   ;
;                      and calls "," (which furnishes the        a --> _       ;
;                      operand for JSR)                                        ;
;                                                                              ;
;    CODE   an "immediate" word (executed even in                              ;
;                      compile mode) which sets the base to hex      _         ;
;                      and sets the code submode flag                          ;
;                                                                              ;
;    ,                 takes a number from the stack and writes  n --> _       ;
;                      it in the dictionary directly                           ;
;                                                                              ;
;    LITERAL           takes a number from the stack and                       ;
;                      generates code which when executed will   n --> _       ;
;                      push the number back on the stack                       ;
;                                                                              ;
;    ;                 closes the current definition by writing                ;
;                      an RTS in the dictionary and clearing the     _         ;
;                      compile and code flags.                                 ;
;                                                                              ;
;    PROMPT            sends prompt (ok) to terminal                 _         ;
;                                                                              ;
;    LINE   gets a line from the input device and            --                ;
;                      places it in the line buffer                            ;
;                                                                              ;
; control structure of the prompt and input code                               ;
;                            PROMPT                                            ;
;                            LINE                                              ;
;                            outer interpreter loop                            ;
;                                   TOKEN                                      ;
;                                   SEARCH                                     ;
;                                   if found                                   ;
;                                      EXECUTE  (execute mode)                 ;
;                                      STKCHK                                  ;
;                                        or                                    ;
;                                      COMPILE  (compile mode)                 ;
;                                   else                                       ;
;                                      NUMBER                                  ;
;                                      if fail                                 ;
;                                           WHAZZAT                            ;
;                                      else                                    ;
;                                           COMMA  (code mode)                 ;
;                                            or                                ;
;                                           LITERAL    (compile mode)          ;
;                                            or                                ;
;                                           return     (execute mode)          ;
;                                                                              ;
; register usage                                                               ;
;                                                                              ;
;    A4      line buffer pointer                                               ;
;    A5      dictionary pointer              D0     I/O port                   ;
;    A6      parameter stack pointer         D1     "scratch                   ;
;    A7      return stack pointer            A0           registers"           ;
;                                                                              ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

          .ABSOLUTE
          .PROC         FLINT
          .ORG          01000H

          BRA           START              ; set up

TERMBUF   .BLOCK        84,32              ;          system buffers,
DISKBUF   .BLOCK        1024,32
          .ASCII        "INTERACTIVE "
          .BYTE         13
          .BYTE         32

RTNSTK    .BLOCK        80,0               ;          system stacks,
PARMSTK   .BLOCK        256,0
UNDRFLW   .WORD         0

DICT      .WORD         LAST               ;          system variables,
CBLOCK    .WORD         0050H
BASE      .WORD         10

FCOLON    .BYTE         0                  ;          system flags,
FIMMED    .BYTE         0
FCODE     .BYTE         0
FNEG      .BYTE         0
FINT      .BYTE         0
          .ALIGN        2
BLANK     .BLOCK        4,32               ;          etc.
ZERO      .BLOCK        3,32
          .BYTE         48

START     LEA           RTNSTK+80,A7       ; initialize pointers
          LEA           PARMSTK+256,A6     ;
          MOVEA.W   BUFPNT,A5              ;
          CLR.L         FCOLON             ; initialize flags
          CLR.B         FINT               ;
          JSR           LOAD               ; load boot screen
RESTART   JSR           WHICHBUF           ; select input buffer

MAIN      JSR           TOKEN              ; get next token
          MOVE.W        DICT,-(A6)         ; push dictionary pointer
          JSR           SEARCH             ; look for token
          TST.W         (A6)+              ; if found then
          BEQ           TSTNUM             ;
          BCLR          #7,FIMMED          ;         if immediate flag off then
```

(continued on next page)

68000 MINI FORTH

Listing Two *(Listing continued, text begins on page 22.)*

```
          BNE       GODO          ;
          TST.B     FCOLON        ;       if in compile mode
          BEQ       GODO          ;
          JSR       COMPILE       ;         compile
          BRA       MAIN          ;   else
GODO      JSR       EXECUTE       ;       execute word
          JSR       STKCHK        ;       and check underflow
          BRA       MAIN          ; else

TSTNUM    MOVE.W    A5,-(A6)      ;   push token buffer address
          JSR       NUMBER        ;   is token a number ??
          TST.W     (A6)+         ;   if not
          BNE       TSTCODE       ;
          JSR       WHAZZAT       ;       ? whazzat ?
          BRA       MAIN          ;   else

TSTCODE   TST.B     FCODE         ;       if code flag on
          BEQ       TSTLIT        ;
          JSR       COMMA         ;          write code in dictionary
          BRA       MAIN          ;       else

TSTLIT    TST.B     FCOLON        ;          if in compile mode
          BEQ       MAIN          ;
          JSR       LITERAL       ;               compile number as literal
          BRA       MAIN          ;

WHICHBUF TST.B    FINT
          BNE       GOLINE        ; if not in interactive mode
          LEA       DISKBUF,A4    ;   get input from disk buffer
          RTS                     ; else

GOLINE    JSR       LINE          ;   fill line buffer from terminal
          RTS                     ;

LINE      JSR       PROMPT        ; prompt
          MOVEQ     #76,D1        ;
          LEA       TERMBUF,A4    ;
CLEANUP   MOVE.L    BLANK,0(A4,D1) ; clear line buffer
          SUBQ.B    #4,D1         ;
          BGE       CLEANUP       ;
          CLR.L     D1            ; clear character count
INCHAR    TGET                    ; get next character and
          CMPI.B    #13,D0        ;   do while not CR
          BEQ       EXIT          ;
          CMPI.B    #8,D0         ;   if character is backspace
          BNE       INBUF         ;
          JSR       RUBOUT        ;       rubout previous char
          BRA       INCHAR        ;   else
INBUF     MOVE.B    D0,0(A4,D1)   ;       copy it into buffer
          ADDQ.B    #1,D1         ;       increment count
          TPUT                    ;       echo to terminal
          BRA       INCHAR        ;
EXIT      MOVE.B    D0,1(A4,D1)   ; imbed CR in buffer
          RTS

RUBOUT    TST.B     D1            ; if count > 0 then
          BLE       RRET          ;
          TPUT                    ;   echo backspace
          SUBQ.B    #1,D1         ;   decrement count
          MOVEQ     #32,D0        ;   erase previous character...
          MOVE.B    D0,0(A4,D1)   ;   in buffer and ...
          TPUT                    ;   on terminal
          MOVEQ     #8,D0         ;
          TPUT                    ;
RRET      RTS                     ; return

STKCHK    LEA       UNDRFLW,A0    ;
          CMPA.W    A0,A6         ; if top is below bottom
          BLE       OKSTK         ;
          JSR       STKU          ;   underflow exception
OKSTK     RTS                     ;

NULLWORD .BLOCK     6,0           ;  DICTIONARY

          .BYTE     4
          .ASCII    "CRL"         ;   CRLF
          .WORD     NULLWORD
CRLF      MOVEQ     #13,D0
          TPUT                    ; send CR
          MOVEQ     #10,D0        ;
          TPUT                    ; send LF
          RTS                     ; return

          .BYTE     6
          .ASCII    "PRO"         ;   PROMPT
          .WORD     CRLF-6
PROMPT    JSR       CRLF          ; send CR LF
          MOVEQ     #111,D0       ;
          TPUT                    ; send "o"
          MOVEQ     #107,D0       ;
          TPUT                    ; send "k"
          MOVEQ     #32,D0        ;
          TPUT                    ; send space
          RTS                     ; return

          .BYTE     7
          .ASCII    "WHA"         ;   WHAZZAT
          .WORD     PROMPT-6
WHAZZAT   JSR       CRLF          ; send CR LF
          MOVE.W    A5,-(A6)      ; push token address
          ADDI.W    #1,(A6)       ;
          MOVE.B    (A5),-(A6)    ; push character count
          CLR.B     -(A6)         ; pad it to "word" length
          JSR       TYPE          ; type offending token
          BRA       RSET
STKU      JSR       CRLF          ; (underflow entry point)
          MOVEQ     #42,D0        ;
          TPUT                    ; send *
```

```
RSET      MOVEQ     #63,D0            ;
          TPUT                        ; send ?
          LEA       PARMSTK+256,A6    ; reset stack pointer
          CLR.L     FCOLON            ; initialize flags
          TAS       FINT              ; set interactive mode
          JSR       CR                ; get new line
          RTS

          .BYTE     5
          .ASCII    "TOK"             ;    TOKEN
          .WORD     WHAZZAT-6
TOKEN     CLR.L     (A5)              ; clear token buffer
          CLR.L     D1                ; clear count
GETCHAR   MOVE.B    (A4)+,D0          ; getcharacter until space
          CMPI.B    #32,D0            ;
          BEQ       EXITGET           ;
          MOVE.B    D0,1(A5,D1)       ;    place in token buffer
          ADDQ.B    #1,D1             ;    increment count
          BRA       GETCHAR           ;
EXITGET   MOVE.B    D1,(A5)           ; put count in 1st byte of buffer
          BEQ       GETCHAR           ; if count is 0 repeat
          RTS                         ; return

          .BYTE     129
          .BYTE     13                ;    (CR)
          .WORD     0
          .WORD     TOKEN-6
CR        LEA       RTNSTK+80,A7      ; reset system stack
          JMP       RESTART           ; restart

          .BYTE     6
          .ASCII    "SEA"             ;    SEARCH
          .WORD     CR-6
SEARCH    MOVE.L    (A5),D1           ; put token "stem" in D1
          MOVEA.W   A6,A0             ; use A0 as search pointer
COMPARE   TST.W     (A0)              ; DO
          MOVEA.W   (A0),A0           ;    get address of next word
          BEQ       NOFIND            ;    if nullword, exit NOFIND
          CMP.L     (A0),D1           ;    compare word to candidate
          BEQ       FIND              ;    if found, exit FIND
          BCHG      #31,D1            ;    set precedence bit
          CMP.L     (A0),D1           ;    compare to "immediate" candidate
          BEQ       FINDIMM           ;    if found, exit FINDIMM
          BCHG      #31,D1            ;    reset precedence bit
          LEA       4(A0),A0          ;    get link address
          BRA       COMPARE           ; LOOP
FINDIMM   TAS       FIMMED            ; set immediate flag
FIND      LEA       6(A0),A0          ; get code address
          MOVE.W    A0,(A6)           ; push it
          MOVE.W    #-1,-(A6)         ; push success flag
          RTS                         ;
NOFIND    MOVE.W    A0,(A6)           ; push fail flag
          RTS

          .BYTE     6
          .ASCII    "NUM"             ;    NUMBER
          .WORD     SEARCH-6
NUMBER    CLR.L     D2                ; clear conversion register
          MOVEA.W   (A6)+,A0          ; get token address
          MOVE.B    (A0)+,D1          ; get digit count
                                      ; DO
NXTDIG    MOVE.B    (A0)+,D0          ;    get next digit
          SUBI.B    #48,D0            ;    strip ASCII prefix
          BLT       FAIL              ;    if digit too small, FAIL
          CMP.W     #10,D0            ;    if digit > 9
          BLT       CMPBASE           ;         adjust for "odd" values
          SUBI.B    #7,D0             ;         and test again
          CMP.W     #10,D0            ;
          BLT       FAIL              ;
CMPBASE   CMP.W     BASE,D0           ;    if base < digit
          BGE       FAIL              ;         FAIL
          MULU      BASE,D2           ;    multiply current value by base
          SWAP      D2                ;
          TST.W     D2                ;    if overflow
          BNE       FAIL              ;         FAIL
          SWAP      D2                ;
          ADD.W     D0,D2             ;    add current digit
          BCS       FAIL              ;    if overflow, FAIL
          SUBQ.B    #1,D1             ;    decrement count
          BNE       NXTDIG            ; UNTIL no digits remain
          MOVE.W    D2,-(A6)          ; push number
          MOVE.W    #-1,-(A6)         ; push success flag
          RTS
FAIL      CLR.W     -(A6)             ; push fail flag
          RTS

          .BYTE     7
          .ASCII    "EXE"             ;    EXECUTE
          .WORD     NUMBER-6
EXECUTE   MOVEA.W   (A6)+,A0          ; pop code address
          JSR       (A0)              ; execute
          RTS

          .BYTE     7
          .ASCII    "COM"             ;    COMPILE
          .WORD     EXECUTE-6
COMPILE   MOVE.W    #04EB8H,(A5)+     ; compile "JSR"
          JSR       COMMA             ; compile code address
          RTS

          .BYTE     6
          .ASCII    "HEA"             ;    HEADER
          .WORD     COMPILE-6
HEADER    MOVE.W    DICT,4(A5)        ; link header to dictionary
          MOVE.W    A5,DICT           ; update DICT
          LEA       6(A5),A5          ; move pointer to code field
          RTS

          .BYTE     9
          .ASCII    "IMM"             ;    IMMEDIATE
          .WORD     HEADER-6
```

(continued on next page)

Listing Two *(Listing continued, text begins on page 22.)*

```
IMMED     MOVEA.W   DICT,A0          ; get address of most recent word
          TAS       (A0)             ; set precedence bit
          RTS

          .BYTE     1
          .ASCII    ":"              ;    ":"
          .WORD     0
          .WORD     IMMED-6
COLON     JSR       TOKEN            ; get token
          JSR       HEADER           ; make header
          TAS       FCOLON           ; set colon flag
          RTS

          .BYTE     132
          .ASCII    "COD"            ;    CODE
          .WORD     COLON-6
CODE      MOVE.W    #16,BASE         ; change BASE to hex
          TAS       FCODE            ; set code flag
          RTS

          .BYTE     129
          .ASCII    ";"              ;    ";"
          .WORD     0
          .WORD     CODE-6
SEMI      MOVE.W    #10,BASE         ; change BASE to decimal
          CLR.B     FCOLON           ; clear colon flag
          CLR.B     FCODE            ; clear code flag
          MOVE.W    #04E75H,(A5)+    ; compile "RTS"
          RTS

          .BYTE     1
          .ASCII    ","              ;    ","
          .WORD     0
          .WORD     SEMI-6
COMMA     MOVE.W    (A6)+,(A5)+      ; pop number to dictionary
          RTS

          .BYTE     7
          .ASCII    "LIT"            ;    LITERAL
          .WORD     COMMA-6
LITERAL   MOVE.W    #03D3CH,(A5)+    ; compile literal code
          JSR       COMMA            ; compile constant
          RTS

          .BYTE     11
          .ASCII    "INT"            ;    INTERACTIVE
          .WORD     LITERAL-6
INTRACTV TAS        FINT             ; set interactive mode
          RTS

          .BYTE     4
          .ASCII    "BAS"            ;    BASE
          .WORD     INTRACTV-6
BASECODE LEA        BASE,A0          ; push BASE address
          MOVE.W    A0,-(A6)
          RTS

          .BYTE     6
          .ASCII    "CBL"            ;    CBLOCK
          .WORD     BASECODE-6
CBLCODE   LEA       CBLOCK,A0        ; push CBLOCK address
          MOVE.W    A0,-(A6)
          RTS

          .BYTE     5
          .ASCII    "EDB"            ;    EDBUF
          .WORD     CBLCODE-6
EDBCODE   LEA       DISKBUF,A0       ; get edit buffer address
          MOVE.W    A0,-(A6)         ; push it
          RTS

          .BYTE     4
          .ASCII    "DIC"            ;    DICT
          .WORD     EDBCODE-6
DICTCODE LEA        DICT,A0          ; get dictionary address
          MOVE.W    A0,-(A6)         ; push it
          RTS

          .BYTE     4
          .ASCII    "LOA"            ;    LOAD
          .WORD     DICTCODE-6
LOAD      GETBLOCK                   ; system dependent macro
          RTS

          .BYTE     2
          .ASCII    "GO"             ;    GO
          .BYTE     0
          .WORD     LOAD-6
GO        CLR.B     FINT             ; leave interactive mode
          JSR       CR               ; restart input sequence

          .BYTE     4
          .ASCII    "SAV"            ;    SAVE
          .WORD     GO-6
SAVE      SAVBLOCK                   ; system dependent macro
          RTS

          .BYTE     4
          .ASCII    "TYP"            ;    TYPE
          .WORD     SAVE-6
TYPE      MOVE.W    (A6)+,D1         ; get character count
          SUBQ.B    #1,D1            ;
          MOVEA.W   (A6)+,A0         ; get buffer address
PUT       MOVE.B    (A0)+,D0         ; DO
          TPUT                       ;    send buffer character
          .WORD     51C9H            ;    to terminal
          .WORD     0FFF6H           ; UNTIL exhausted
          RTS
```

(continued on page 58)

Listing Two *(Listing continued, text begins on page 22.)*

```
         .BYTE       1
         .ASCII      "."                        ;     "."
         .WORD       0
         .WORD       TYPE-6
PRINT    MOVEQ       #13,D0                     ; send CR
         TPUT                                   ;
         MOVEQ       #10,D0
         TPUT
         MOVEA.W  A5,A0                         ; get buffer pointer
         MOVE.L      BLANK,(A0)+                ;
         MOVE.L      BLANK,(A0)+                ; zero out buffer
         MOVE.L      BLANK,(A0)+                ;
         MOVE.L      ZERO,(A0)+                 ;
         MOVE.W      (A6)+,D2                   ; pop number
         BGE         DLOOP                      ; if negative
         NEG.W       D2                         ;    make positive
         TAS         FNEG                       ;    set negative flag
DLOOP    ANDI.L      #65535,D2                  ; while quotient not 0 do
         BEQ         TSTMINUS                   ;    clear remainder
DIV      DIVS        BASE,D2                    ;    divide by base
         MOVE.L      D2,D0                      ;    (do dirty work in D0)
         SWAP        D0                         ;    get remainder
         CMPI.B      #10,D0                     ;
         BLT         PREFIX                     ;    convert to digit
         ADDQ.B      #7,D0                      ;
PREFIX   ADDI.B      #48,D0                     ;    make ASCII prefix
         MOVE.B      D0,-(A0)                   ;    place digit in buffer
         BRA         DLOOP                      ;
TSTMINUS BCLR        #7,FNEG                    ; test and clear negative flag
         BEQ         PRNT                       ; if set
         MOVE.B      #45,-(A0)                  ;    put minus sign in buffer
PRNT     MOVE.W      A5,-(A6)                   ; push buffer address
         MOVE.W      #16,-(A6)                  ; push buffer length
         JSR         TYPE                       ; type number
         RTS

         .BYTE       2
         .ASCII      ".S"                       ;     .S
         .BYTE       0
         .WORD       PRINT-6
SPRINT   LEA         UNDRFLW,A1                 ; get address of bottom
         MOVEA.W  A6,A2                         ;
BOTTOM   CMPA.W      A1,A2                      ; while above bottom do
         BEQ         DONE                       ;
         MOVE.W      (A2)+,-(A6)                ;    push next number
         JSR         PRINT                      ;    print it
         BRA         BOTTOM                     ;
DONE     RTS

         .BYTE       129
         .ASCII      "("                        ;     "("
         .WORD       0
         .WORD       SPRINT-6
CMMNT    MOVE.B      (A4)+,D0                   ; get character from input buffer
         CMPI.B      #41,D0                     ; until ")"
         BNE         CMMNT                      ;
         RTS

         .BYTE       4
         .ASCII      "QUI"                      ;     QUIT
         .WORD       CMMNT-6
QUIT     XSTOP                                  ; return to monitor
         RTS

LAST     .BYTE       6
         .ASCII      "LOG"                      ;     LOGOFF
         .WORD       QUIT-6
         LEA         BUFPNT,A0                  ; save dictionary pointer
         MOVE.W      A5,(A0)
         RTS

BUFPNT   .WORD       BUFFER
BUFFER   .WORD
         .END
```

End Listing Two

Listing Three

```
(Listing Three)
("inner shell" words for FLINT)

(The "->" symbol when used in a comment signifies that the)
(instruction corresponding to the preceding assembler)
(mnemonic will be written in the dictionary at execution time.)

: CONSTANT   ( n --                creates a constant)
  TOKEN HEADER LITERAL CODE 3AFC 4E75  ( rts  -> ) ;

: CREATE     ( --        creates the header and code body for a variable)
  TOKEN HEADER CODE 2AFC 41FA 0006  ( lea      6[a5],a0        -> )
                         3AFC 3D08          ( move.w   a0,-[a6]         -> )
                         3AFC 4E75          ( rts  -> ) ;

: ALLOT          ( n -- _           used after CREATE to allocate space for a )
                 (                  variable or an array                        )
  CODE DADE  ( ADDA.W          [A6]+,A5   ) ;

: VARIABLE   ( --        creates a variable)
  CREATE 2 ALLOT ;

: @              ( a -- n           "fetch" - replaces an address with its value)
  CODE 305E  ( MOVEA.W         [A6]+,A0   )
       3D10  ( MOVE.W          [A0],-[A6] ) ;
```

```
: !              ( n a -- _          stores a word length value in the address)
  CODE 305E  ( MOVEA.W               [A6]+,A0   )
       309E  ( MOVE.W                [A6]+,[A0] ) ;

: !BYTE          ( n a -- _          stores a byte length value in the address)
  CODE 305E  ( MOVEA.W              [A6]+,A0   )
       4A1E  ( TST.B [A6]+                     )
       109E  ( MOVE.B               [A6]+,[A0] ) ;

: HEX            ( --      changes the system base to 16)
  16 BASE ! ;

: DECIMAL      ( --        changes the system base to 10)
  10 BASE ! ;

: SWAP           ( n1 n2 -- n2 n1                   )
  CODE 221E  ( MOVE.L             [A6]+,D1 )
       4841  ( SWAP  D1                    )
       2D01  ( MOVE.L             D1,-[A6] ) ;

: DUP            ( n -- n n )
  CODE 3D16  ( MOVE.W              [A6],-[A6] ) ;

: ?              ( n -- _            tests the top value, drops it, and sets CCs)
  CODE 4A5E  ( TST.W [A6]+         ) ;

: \              ( n -- _            synonym for "?"   used to emphasize the drop)
  CODE 4A5E  ( TST.W [A6]+         ) ;

: OVER           ( n1 n2 -- n1 n2 n1 )
  CODE 3D2E 0002  ( MOVE.W   2[A6],-[A6] ) ;

: 2DUP           ( n1 n2 -- n1 n2 n1 n2 )
  OVER OVER ;

: >R             ( n -- _            removes a value from the parameter stack )
                 (                   and places it on the return stack         )
  CODE 221F  ( MOVE.L            [A7]+,D1 )
       3F1E  ( MOVE.W            [A6]+,-[A7] )
       2F01  ( MOVE.L            D1,-[A7] ) ;

: <R             ( _ -- n            removes a value from the return stack     )
                 (                   and places it on the parameter stack      )
  CODE 221F  ( MOVE.L            [A7]+,D1 )
       3D1F  ( MOVE.W            [A7]+,-[A6] )
       2F01  ( MOVE.L            D1,-[A7] ) ;

: ROT            ( n1 n2 n3 -- n2 n3 n1 )
  >R SWAP <R SWAP ;

: +              ( n1 n2 -- n1+n2 )
  CODE 321E  ( MOVE.W              [A6]+,D1 )
       D356  ( ADD.W D1,[A6]     ) ;

: ~              ( n -- -n )
  CODE 4456  ( NEG.W [A6] ) ;

: -              ( n1 n2 -- n1-n2 )
  ~ + ;

: *              ( n1 n2 -- n1*n2 )
  CODE 321E  ( MOVE.W              [A6]+,D1 )
       C3DE  ( MULS  [A6]+,D1 )
       3D01  ( MOVE.W              D1,-[A6] ) ;

: /MOD           ( n1 n2 -- n1/n2 n1 mod n2 )
  CODE 321E  ( MOVE.W              [A6]+,D1 )
       341E  ( MOVE.W              [A6]+,D2 )
       48C2  ( EXT.L D2                     )
       85C1  ( DIVS  D1,D2                  )
       3D02  ( MOVE.W              D2,-[A6] )
       4842  ( SWAP  D2                     )
       3D02  ( MOVE.W              D2,-[A6] ) ;

: /              ( n1 n2 -- n1/n2 )
  /MOD \ ;

: MOD            ( n1 n2 -- n1 mod n2 )
  /MOD SWAP \ ;

: 0<             ( n -- f        returns a true flag if n < 0)
  CODE 4A56               ( TST.W [A6]          )
       6D04               ( BLT   4[PC]         )
       4256               ( CLR.W [A6]          )
       4E75               ( RTS                 )
       3CBC FFFF    ( MOVE.W    #-1,[A6] ) ;

: 0>             ( n -- f        returns a true flag if n > 0)
  ~ 0< ;

: <              ( n1 n2 -- f   returns a true flag if next < top)
  - 0< ;

: >              ( n1 n2 -- f   returns a true flag if next > top)
  - 0> ;

: CGET           ( _ -- n            gets a character from the terminal and)
                 (                   places its ASCII value on the stack     )
  CODE 4EB9 00FE 0008  ( TGET                    )
       3D00                ( MOVE.W        D0,-[A6] ) ;

: EMIT           ( n -- _            takes an ASCII value from the stack and)
                 (                   sends it to the terminal                 )
  CODE 301E                ( MOVE.W        [A6]+,D0 )
       4EB9 00FE 0008  ( TPUT                 ) ;

: CLEAR          ( --      erases the screen)
  26 EMIT ;

: ->             ( --      increments CBLOCK; loads and executes )
                 (                   the new block [allows chaining]        )
```

(continued on page 61)

PAINLESS WINDOWS.

Windows. Data Entry. Menus. Finally, a C programmers' tool that makes them as easy to use as *printf()*. With Greenleaf DataWindows™, you move in quantum leaps!

Snazzy Window Treatments

DataWindows represents an important breakthrough in C programming tools. It sets you free so you can create exciting programs quickly and easily, saving both time and money! Developed to work with the IBM PC, XT, AT, compatibles, and MSDOS or PCDOS, DataWindows is a carefully tooled system of C functions which will jazz up your programs with unprecedented efficiency.

Greenleaf DataWindows is integrated windows, transaction data entry, pop-up, pull-down, and Lotus style menu systems with:

- **Screen Management.** You don't have to remember what's on the display or the sequence in which you put it there. DataWindows does the grunt work. There are no restrictions.
- **Transaction Data Entry.** Data entry windows can have any number of fields with sophisticated options for reading many data types. Calls are made to help, validation, and other functions. Full featured text editing, protected and mandatory fields, dBASE type picture strings, context sensitive help, validation of fields and transactions, redefinable keys, password entry, attribute control, keyboard idle and much more.
- **Device Independence.** It detects the type of display adapter your computer is using and adjusts to it automatically for CGA, EGA, or monochrome. Logical video attributes are easy to use for color or monochrome.
- **Compatibility.** Runs with Microsoft Windows and IBM TopView.
- **The Greenleaf Tradition of Quality.** Reliable products. Professional documentation that gets you up and running quickly and keeps you there. Reference card. Newsletter and Bulletin board.

IBM, Microsoft & dBase, are registered trademarks of International Business Machines, Microsoft Corporation & Ashton-Tate respectively. PCDOS, IBM PC, XT, AT, & TopView are trademarks of IBM; MSDOS and Microsoft Windows are trademarks of Microsoft Corporation.

Stop Window Shopping

Order Today. Or call toll free for a free demo of the windows library that makes all the others obsolete.

Order any of these high performance tools by calling your dealer or 1-800-523-9830 today. Specify compiler when ordering. Add $8 for UPS second day air, or $5 for ground. Texas residents add sales tax. MasterCard, VISA, P.O., check, COD. In stock, shipped next day.

Greenleaf DataWindows	$225
DataWindows Source Module	$225
The Greenleaf Comm Library v2.0	$185
The Greenleaf Functions v3.0	$185
Digiboard Comm/4-II	$325
Digiboard Comm/8-II	$535

GREENLEAF Software©

1411 LeMay Drive, Suite 101
Carrollton, TX 75007

Call Toll Free
1-800-523-9830
In Texas and Alaska, call
214-446-8641

Circle **no. 97** on reader service card.

Window Dressings

- **Simple or Complex Windows.** Up to 254 powerful overlaid windows simultaneously, all with just one kind of window to remember! Yet any window can be from one character to 32K!
- **Easy Window Operations.** DataWindows lets you move, zoom, frame, title, change colors, titles, frames, size, location, and make windows visible or invisible at will! Functions set cursor, attributes, and write data to any window or "current window". Word wrap, auto scroll, keyboard functions.
- **Write to Any Window Any Time.** Windows may be visible, overlaid, or invisible, and you can write to them anyway. What you write will be seen when the windows become visible.
- **DataWindows is fast!** It writes directly to video memory (in some modes).
- **Easy to save!** Any window, complete with attributes, can be saved on disk quickly and efficiently.
- **Source code available. No royalties.**

Also from Greenleaf:

The Greenleaf Functions v3.0
The most complete, mature C language function library for the IBM PC,XT,AT and close compatibles. Includes over 225 functions — DOS, disk, video, color text and graphics, string, time/date, keyboard, disk status and Ctrl-Break functions plus many more.

The Greenleaf Comm Library
Our 2.0 version is the hottest communications facility of its kind. Over 120 functions — ring buffered, interrupt driven asynchronous communications for up to 16 ports simultaneously with XMODEM, XON/XOFF, many many sophisticated features.

We support all popular C compilers for MSDOS/PCDOS: Microsoft, Lattice, Computer Innovations, Aztec, DeSmet, and others.

Listing Three *(Listing continued, text begins on page 22.)*

```
  CBLOCK @  2 +       CBLOCK !  LOAD       GO ;

: ?>             ( -- )
  CODE 3AFC 4A5E  ( tst.w    [a6]+  -> ) ;

: BRA>           ( -- )
  CODE 3AFC 6000  ( bra            -> ) ;

: BEQ>           ( -- )
  CODE 3AFC 6700  ( beq            -> ) ;

: BNE>           ( -- )
  CODE 3AFC 6600  ( bne            -> ) ;

: MARK           (      _ -- a          pushes the contents of the dictionary pointer)
  CODE 3D0D  ( MOVE.W          A5,-[A6] ) ;

: SPLIT          (      _ -- a )
  ?> BEQ> MARK 2 ALLOT ;

: JOIN           ( -- )
  DUP MARK SWAP - SWAP ! ;

: IF             ( :  _ -- a   x  f -- _ )
  SPLIT ; IMMEDIATE

: THEN           ( :  a -- _   x  -- )
  JOIN ; IMMEDIATE

: ELSE           ( :  a1 -- a2   x         -- )
  BRA> MARK 2 ALLOT SWAP JOIN ; IMMEDIATE

: DO             ( :  _ -- a   x  -- )
  MARK ; IMMEDIATE

: UNTIL          ( :  a -- _   x  f -- _ )
  ?> BNE> MARK - , ; IMMEDIATE

: WHILE          ( :  _ -- a   x  f -- _ )
  SPLIT ; IMMEDIATE

: LOOP           ( :  a1 a2 -- _   x  -- )
  BRA> SWAP MARK - , JOIN ; IMMEDIATE

: '              ( _ -- a         pushes the address of the token which follows ')
  TOKEN DICT @ SEARCH IF ELSE WHAZZAT THEN ;

: FORGET     ( --     erases the dictionary entries for the token following)
                 (                FORGET as well as all tokens which succeed it in the )
                 (                dictionary                                            )
  ' DUP 2 - @  DICT !  6 - CODE 3A5E ( MOVEA.W      (A6)+,A5 ) ;
```

End Listings

How to tackle a 300 page monster.

Turn your PC into a typesetter.

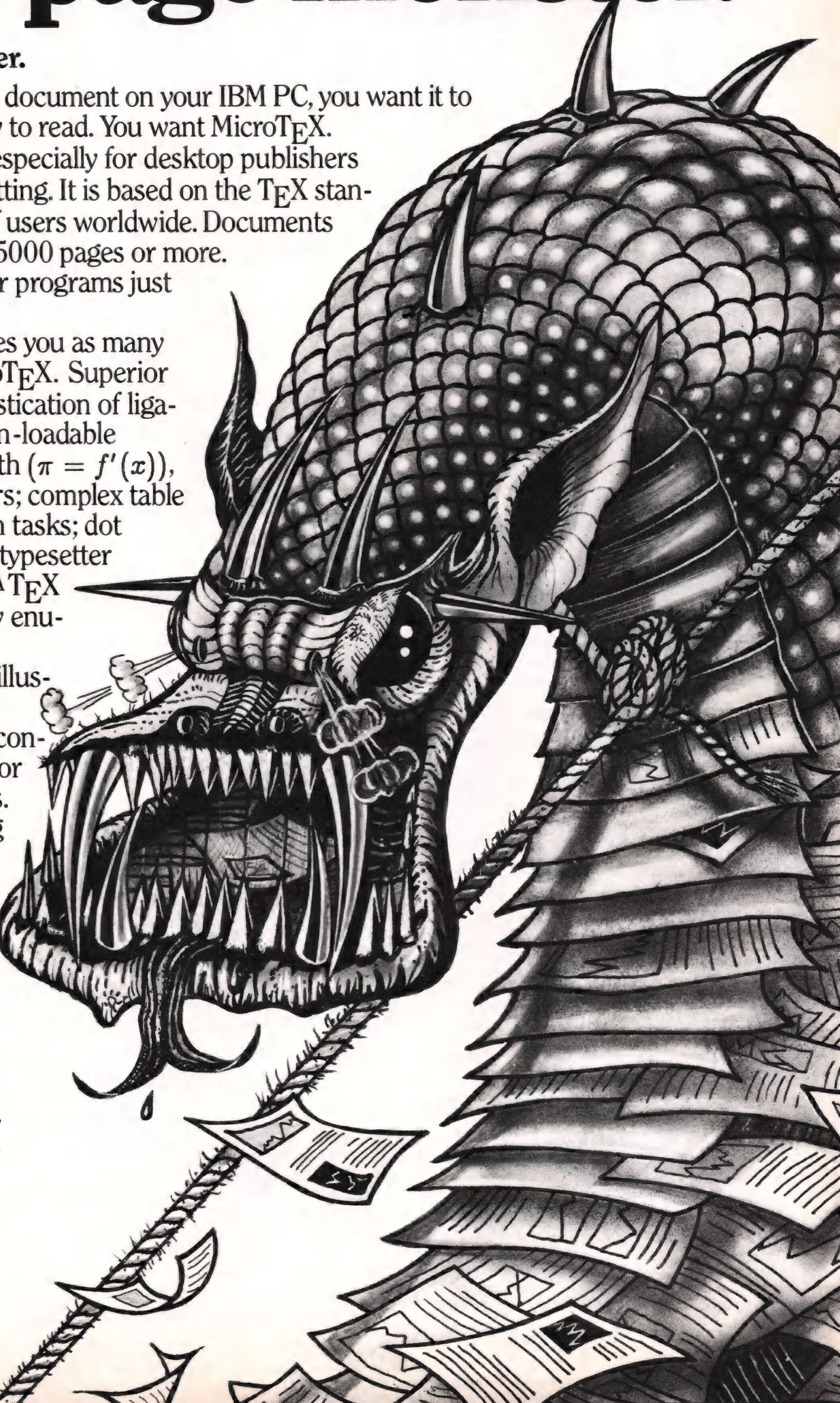
If you're writing a long, serious document on your IBM PC, you want it to look professional. Precise. Easy to read. You want MicroTeX.

MicroTeX was designed especially for desktop publishers who require heavy duty typesetting. It is based on the TeX standard, with tens of thousands of users worldwide. Documents from smaller than 30 pages to 5000 pages or more. And that's something that other programs just can't match.

No other PC software gives you as many advanced capabilities as MicroTeX. Superior hyphenation control, the sophistication of ligatures (ffi, fi) and kerning; down-loadable fonts; aesthetic handling of math ($\pi = f'(x)$), and foreign language characters; complex table construction and multi-column tasks; dot matrix, laser printer and phototypesetter output. When used with our LaTeX macro package, it automatically enumerates and cross-references pages, sections, footnotes and illustrations. Plus it automatically creates your indexes, tables of contents, and even updates them for you after last minute insertions.

So if you want typesetting software that's as serious as you are about your writing, get MicroTeX. ***Call toll free 800-255-2550*** or 617-944-6795 to order or for more information.* Order with a 60-day money back guarantee.

Now available for Macintosh.



MicroTeX™
from Addison-Wesley

Serious typesetting for serious desktop publishers.

*Dealers, call our Dealer Hot Line: 800-447-2226 (In MA, 800-446-3399), ext. 2643.

Circle **no. 92** on reader service card.

Listing One *(Text begins on page 40.)*

```
Listing One

              Include   MacTraps.D
              Include   ToolEqu.D
              Include   SysEqu.D
              Include   QuickEqu.D

              PEA       -4(A5)                    ; initialize managers
              _InitGraf
              _InitFonts
              MOVE.L    #$0000FFFF,D0
              _FlushEvents
              _InitWindows
              _InitMenus
              CLR.L     -(SP)
              _InitDialogs
              _InitCursor

              CLR       -(SP)
              PEA       'DrD.Rsrc'
              _OpenResFile                        ;open resource file
              MOVE      (SP)+,D0                  ;discard unused result
;--------------------------- Set up menus ------------------------------
              CLR.L     -(SP)                     ;space for handle
              MOVE      #1,-(SP)                  ;menu ID
              _GetRMenu                           ;get Apple menu template
              MOVE.L    (SP)+,AppleHandle(A5)     ;retrieve & store handle

              MOVE.L    AppleHandle(A5),-(SP)     ;put handle back on stack
              MOVE.L    #'DRVR',-(SP)             ;res type for desk accs
              _AddResMenu                         ;get desk accessories

              MOVE.L    AppleHandle(A5),-(SP)
              CLR       -(SP)                     ;put menu after all others
              _InsertMenu                         ;put menu in menu list

              CLR.L     -(SP)                     ;repeat for other menus
              MOVE      #2,-(SP)
              _GetRMenu
              MOVE.L    (SP)+,FileHandle(A5)

              MOVE.L    FileHandle(A5),-(SP)
              CLR       -(SP)
              _InsertMenu

              CLR.L     -(SP)
              MOVE      #3,-(SP)
              _GetRMenu
              MOVE.L    (SP)+,EditHandle(A5)

              MOVE.L    EditHandle(A5),-(SP)
              CLR       -(SP)
              _InsertMenu

              _DrawMenuBar

; ------------------ Open the window with a text edit record -------------
              CLR.L     -(SP)                     ;space for window pointer
              MOVE      #1,-(SP)                  ;window ID
              PEA       WindowStorage(A5)         ;window storage
              MOVE.L    #-1,-(SP)                 ;put window in front
              _GetNewWindow
              MOVE.L    (SP)+,WindowPtr(A5)

              MOVE.L    WindowPtr(A5),-(SP)
              _SetPort                            ;makes window current grafport

              CLR.L     -(SP)                     ;place for text handle
              PEA       DestRect                  ;destination rectangle
              PEA       ViewRect                  ;view rectangle
              _TENew                              ;establish text edit record
              MOVE.L    (SP)+,TextHandle(A5)      ;get handle

              MOVE.L    TextHandle(A5),-(SP)
              _TEActivate                         ;make text edit record active
; ------------------ Event loop begins here ------------------------------
        Event
              _SystemTask                         ;update desk accessories
              MOVE.L    TextHandle(A5),-(SP)
              _TEIdle                             ;make cursor blink

              CLR       -(SP)                     ;space for boolean result
              MOVE      #-1,-(SP)                 ;mask to select all events
              PEA       EventRecord(A5)           ;pointer to event record
              _GetNextEvent                       ;get event from queue

              MOVE      (SP)+,D0                  ;retrieve boolean result
              BEQ       Event                     ;no event

              MOVE      EventRecord(A5),D0        ;get event type
```

(continued on page 66)

Listing One *(Listing continued, text begins on page 40.)*

```
                CMP     #mButDwnEvt,D0              ;mouse event?
                BEQ     MouseEvent

                CMP     #keyDwnEvt,D0               ;key pressed?
                BEQ     KeyEvent

                CMP     #upDatEvt,D0
                BEQ     Update                      ;refresh?

                BRA     Event                       ;no desired event posted

;------------------------ Handle key down events -------------------------
        KeyEvent
                MOVE    EventRecord+evtMeta(A5),D0
                BTST    #cmdKey,D0                  ;command key pressed?
                BNE     KeyboardEquivalent

                MOVE    EventRecord+evtMessage+2(A5),-(SP)  ;character pressed
                MOVE.L  TextHandle(A5),-(SP)
                _TEKey                              ;insert character

                BRA     Event

        KeyboardEquivalent
                CLR.L   -(SP)                       ;place for menu ID & item nubmer
                MOVE    EventRecord+evtMessage+2(A5),-(SP)  ;character
                _MenuKey                            ;which menu?

                BRA     Selections

;------------------------- Update the text window ------------------------
        Update
                MOVE.L  WindowPtr(A5),-(SP)
                _BeginUpdate

                MOVE.L  WindowPtr(A5),-(SP)
                _SetPort

                PEA     ViewRect
                MOVE.L  TextHandle(A5),-(SP)
                _TEUpdate

                MOVE.L  WindowPtr(A5),-(SP)
                _EndUpdate

                BRA     Event

;------------------------- Handle mouse down events ----------------------
        MouseEvent
                CLR     -(SP)                       ;space for "what" result
                MOVE.L  EventRecord+evtMouse(A5),-(SP) ;place where event occurred
                PEA     WhichWindowPtr(A5)  ;window affected goes here
                _FindWindow                         ;get exact location of event
                MOVE     (SP)+,D0                   ;recover result

                CMP     #inMenuBar,D0               ;in menu bar?
                BEQ     MenuBar

                CMP     #inSysWindow,D0             ;in desk accessory?
                BEQ     SysEvent

                CMP     #inContent,D0               ;in text edit area?
                BEQ     ApplWindow

                CMP     #inGoAway,D0                ;in close box?
                BEQ     GoAwayBox

                BRA     Event                       ;not an event this program handles

;--------------- Handle events in system windows -------------------
        SysEvent
                PEA     EventRecord(A5)
                MOVE.L  WhichWindowPtr(A5),-(SP)    ;window posting event
                _SystemClick                        ;let system handle it

                BRA     Event

;------------------- Handle events in content area of window -----------
        ApplWindow
                PEA     EventRecord+evtMouse(A5)    ;event location
                _GlobalToLocal                      ;convert coordinates

                MOVE.L  EventRecord+evtMouse(A5),-(SP)  ;coordinates now local
                MOVE    EventRecord+evtMeta(A5),D0
                BTST.L  #shiftKey,D0                ;extended selection?
                SNE     D0
                MOVE.B  D0,-(SP)                    ;extend or not extend
                MOVE.L  TextHandle(A5),-(SP)
                _TEClick                            ;set selection range

                BRA     Event
```

(continued on page 68)

Circle **no. 173** on reader service card.

Listing One *(Listing continued, text begins on page 40.)*

```
;---------------- Handle events in menu bar -----------------------
        MenuBar
                CLR.L     -(SP)                          ;space for menu ID & item
                MOVE.L    EventRecord+evtMouse(A5),-(SP) ;place where event occurred
                _MenuSelect                              ;get menu ID & item

        Selections
                MOVE.L    (SP)+,D7  ;recover result
                MOVE      D7,D6                ;D6 now has menu item
                SWAP      D7                   ;low-order word has menu ID

                CLR       -(SP)                ;selects all menus
                _HiLIteMenu                    ;remove highlighting

                CMP       #1,D7                ;apple menu?
                BEQ       AppleMenu

                CMP       #2,D7                ;file menu?
                BEQ       FileMenu

                CMP       #3,D7                ;edit menu?
                BEQ       EditMenu

                BRA       Event

;------------- Handle desk accessories ----------------------------
        AppleMenu
                MOVE.L    AppleHandle(A5),-(SP)
                MOVE      D6,-(SP)  ;menu item
                PEA       DeskAccName(A5)      ;space for desk accessory name
                _GetItem

                CLR       -(SP)                ;space for reference number
                PEA       DeskAccName(A5)      ;desk accessory name
                _OpenDeskAcc                   ;open the desk accessory
                MOVE      (SP)+,D0             ;discard result

                BRA       Event

;------------------- Handle editing --------------------------------
        EditMenu
                SUBQ      #1,D6                ;adjust item selected for SysEdit
                CLR       -(SP)                ;space for result
                MOVE      D6,-(SP)             ;adjusted item number
                _SysEdit

                MOVE      (SP)+,D0             ;get result
                BNE       Event                ;system handled edit

                ADDQ      #1,D6                ;restore item nuamber
                CMP       #3,D6                ;cut?
                BNE       EditMenu2
                MOVE.L    TextHandle(A5),-(SP)
                _TECut
                BRA       Event

        EditMenu2
                CMP       #4,D6                ;copy?
                BNE       EditMenu3
                MOVE.L    TextHandle(A5),-(SP)
                _TECopy
                BRA       Event

        EditMenu3
                CMP       #5,D6                ;paste?
                BNE       EditMenu4
                MOVE.L    TextHandle(A5),-(SP)
                _TEPaste
                BRA       Event

        EditMenu4
                CMP       #6,D6                ;clear?
                BNE       Event
                MOVE.L    TextHandle(A5),-(SP)
                _TEDelete
                BRA       Event

;--------------- Handle file menu ---------------------------------
        FileMenu
                CMP       #2,D6                ;close selected?
                BEQ       CloseAndQuit

                CMP       #4,D6
                BEQ       CloseAndQuit         ;quit selected

;!!!!!all other file menu options are not implemented in this program!!!!

                BRA       Event

;---------------- Close the window and quit ---------------------
        GoAwayBox
```

```
            CLR.B      -(SP)                        ;space for boolean result
            MOVE.L     WhichWindowPtr(A5),-(SP)     ;window pointer
            MOVE.L     EventRecord+evtMouse(A5),-(SP)      ;point of event
            _TrackGoAway                   ;monitor GoAway box

            MOVE.B     (SP)+,D0  ;get result
            BEQ        Event               ;don't close

    CloseAndQuit
            MOVE.L     TextHandle(A5),-(SP)
            _TEDispose                     ;close text edit record

            MOVE.L     WindowPtr(A5),-(SP)
            _CloseWindow                   ;close the window

            RTS                            ;return to Finder

;--------------------- Data structures ---------------------------
    AppleHandle        DS.L     1
    FileHandle         DS.L     1
    EditHandle         DS.L     1

    WindowPtr DS.L     1
    WindowStorage      DS       WindowSize

    TextHandle         DS.L     1
    ViewRect  DC       3,3,300,490
    DestRect  DC       3,3,300,490

    EventRecord        DS.B     16
    WhichWindowPtr     DS.L     1
    DeskAccName        DS       16
```

End Listing One

Listing Two

```
Listing Two

        DrD.Rsrc

        TYPE MENU ;menu templates follow
         ,1       ;resource ID
        \14       ;will create Apple icon for title

         ,2       ;resource ID
        File      ;menu title
        New/N     ;all the rest are menu items
        Open/O
        Close/W
        Save As...
        Save/S
        Page Setup...
        Print/P
        Quit/Q

         ,3       ;resource ID
        Edit      ;menu title
        Undo/Z
        (-        ;straight line - disabled
        Cut/X
        Copy/C
        Paste/V
        Clear

        TYPE WIND           ;window templates follow
         ,1                 ;resource ID
        Dr. Dobb's Journal  ;window title
        50 10 310 502       ;initial coordinates
        Visible GoAway      ;make window visible, draw GoAway box
        0                   ;window type (standard document window)
        0                   ;optional reference number
```

End Listing Two

Listing Three

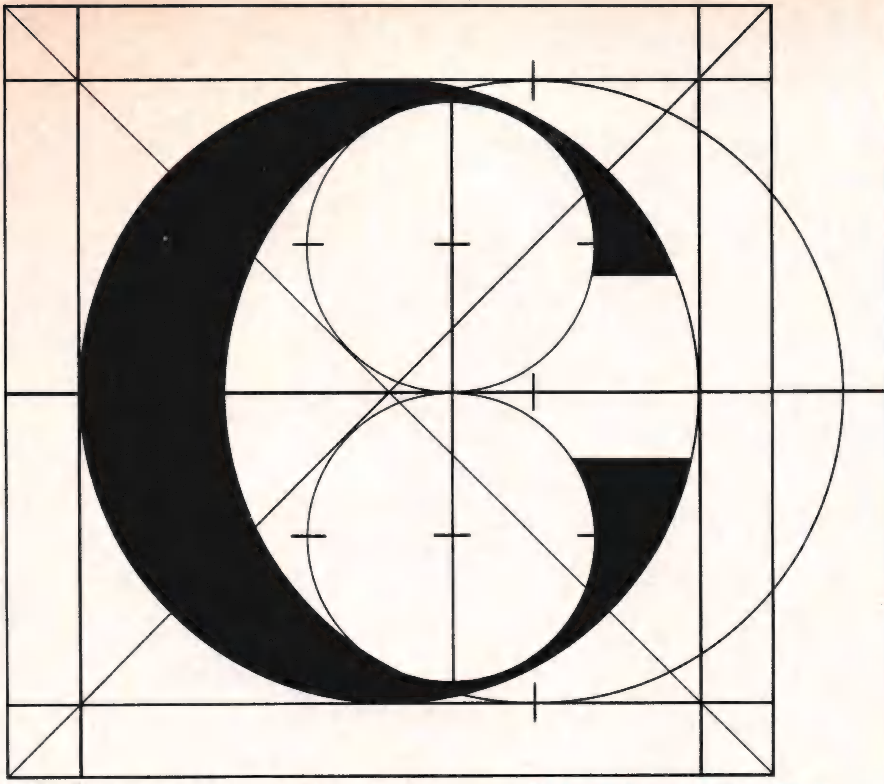
```
Listing Three

                include   "exec/types.i"
                include   "exec/exec.i"
                include   "intuition/intuition.i"

        callsys macro
                CALLIB    _LVO\1    ;calls a system routine
                endm

        xlib    macro
                xref      _LVO\1    ;for library routines
                endm
```

(continued on page 70)

Listing Three *(Listing continued, text begins on page 40.)*

```
        passtext macro
                lea       \1,A0     ;pointer to text
                lea       \2,A1     ;ptr to Intuition text structure
                jsr       SetText   ;initializes text structure
                endm

        passitem macro
                lea       \1,A0     ;pointer to menu item structure
                move.l    \2,A1     ;pointer to next menu item in list
                lea       \3,A2     ;pointer to text structure
                move.b    \4,D0     ;keyboard equivalent
                move      \5,D1     ;offset from top of menu item box
                jsr       SetItem   ;initializes menu item structure
                endm

                xlib      AllocSignal ;external refs for all system
                xlib      AllocMem  ;routines that the program will call
                xlib      FreeSignal
                xlib      AddPort
                xlib      NewList
                xlib      FindTask
                xlib      OpenLibrary
                xlib      OpenWindow
                xlib      SetMenuStrip
                xlib      OpenDevice
                xlib      DoIO
                xlib      SendIO
                xlib      Wait
                xlib      GetMsg
                xlib      ReplyMsg
                xlib      CloseDevice
                xlib      CloseWindow
                xlib      CloseScreen
                xref      _AbsExecBase ;exec's base is fixed

        FrontPen equ      0
        BackPen  equ      1         ;for rendering window and text

;---------------- Open Intuition library ----------------------------
                move.l    _AbsExecBase,A6
                lea       IntName,A1          ;name of library to open
                move.l    #0,D0
                callsys   OpenLibrary
                bne       Continue
                rts                 ;unsuccessful opening ends program

        Continue clr.l    IntBase
                move.l    D0,IntBase          ;save base of Intuition library

;------------------- Create a custom screen -------------------------
                lea       TheScreen,A0        ;pointer to screen data structure
                move      #0,ns_LeftEdge(A0) ;coordinates of screen
                move      #0,ns_TopEdge(A0)
                move      #320,ns_Width(A0)
                move      #200,ns_Height(A0)
                move      #2,ns_Depth(A0)     ;graphics depth
                move.b    #0,ns_DetailPen(A0) ;color for details
                move.b    #1,ns_BlockPen(A0)  ;color for rest of drawing
                move      #0,ns_ViewModes(A0)
                move      #CUSTOMSCREEN,ns_Type(A0)
                move.l    #0,ns_Fonts(A0)     ;use default font
                lea       ScreenTitle,A1
                move.l    A1,ns_DefaultTitle(A0)
                move.l    #0,ns_Gadgets(A0)   ;no special gadgets attached
                move.l    IntBase,A6
                callsys   OpenScreen
                move.l    D0,ScreenPtr        ;results almost always come back in D0

;----------------------- Open a window ------------------------------
                lea       TheWindow,A0        ;pointer to window data structure
                move      #20,nw_LeftEdge(A0)           ;initial coordinates
                move      #20,nw_TopEdge(A0)
                move.b    #0,nw_DetailPen(A0)           ;color for characters
                move.b    #1,nw_BlockPen(A0)  ;color for rest of drawing
                lea       WindowTitle,A1
                move.l    A1,nw_Title(A0)
                move.l    #WINDOWCLOSE+SMART_REFRESH+ACTIVATE+WINDOWDRAG+
                          WINDOWSIZING+WINDOWDEPTH,nw_Flags(A0)
                                              ;system gadgets, etc.
                move.l    #CLOSEWINDOW+MENUPICK,nw_IDCMPFlags(A0)
                                              ;events to be reported
                move      #CUSTOMSCREEN,nw_Type(A0)
                move.l    #0,nw_FirstGadget(A0)  ;no special gadgets attached
                move.l    #0,nw_CheckMark(A0) ;not using checked menu items
                move      #150,nw_Height(A0)  ;initial
                move      #280,nw_Width(A0)   ;initial
                move      #100,nw_MinWidth(A0) ;since window can be sized
                move      #25,nw_MinHeight(A0)
                move      #640,nw_MaxWidth(A0)
                move      #200,nw_MaxHeight(A0)
                move.l    ScreenPtr,nw_Screen(A0)
                move.l    IntBase,A6
```

(continued on page 72)



WRITE BETTER C PROGRAMS.

C TOOLS PLUS

Programmer's Connection is pleased to present *C TOOLS PLUS*, one of the most complete C language function libraries available today. It combines the general purpose capabilities of C TOOLS and C TOOLS 2 (two packages that receive rave reviews for quality, organization, usability and documentation) with extensive new windowing capabilities.

Windows

For the first time, you can get full functioned window support in a general purpose library. The routines are easy to use for pop-up menus or windows and can give you some very advanced capabilities. Windows can be stacked, removed and can accept user input. There is no limit, other than available dynamic memory, to the number of windows you can construct and use.

Video Support

C TOOLS PLUS has fast direct video access for efficiency that will not constrain good program design. You can write directly to the video adapter for efficiency or use only BIOS calls if you're working in a TopView-like environment. It fully supports EGA text modes including 43-line mode and multiple display pages.

Memory-resident DOS Access

Blaise Computing's C TOOLS 2 package is well-known for effectively allowing you to write interrupt service routines in C. *C TOOLS PLUS* takes these capabilities even further. It demonstrates how to access DOS functions from within an interrupt service routine, so now you can write memory-resident routines that access the disk.

General Purpose Functions

C TOOLS PLUS consists of over 200 carefully written functions that are easy-to-use and supplied in source code and pre-built library form. These tools isolate hardware dependence, are small and are written predominently in C.

Some of the areas covered are: • extensive string handling • screen handling • graphics interface • general utility and keyboard functions • DOS memory management • external program and DOS internal command execution • DOS file handling and directory maintenance • and much more.

All Blaise Computing products come with complete, fully documented source code as well as a complete comprehensive, indexed reference manual.

Requirements

C TOOLS PLUS requires an IBM personal computer (PC/XT/AT) or compatible, although many of the functions require only MS—DOS Version 2.00 or later. *C TOOLS PLUS* supports the Microsoft and Lattice C compliers (versions 3.00 or later).

Other Products

Other quality products for C from Blaise Computing include:

ASYNCH MANAGER — provides the crucial core of hardware interrupt support needed to build applications that need asynchronous communication facilities. It also includes routines for XMODEM file-transfer protocol and support for Hayes-compatible modems.

VIEW MANAGER — a development system for building data entry screens and menus. Begin by designing on-screen what the operator will see, then call upon library functions from your program to display the screens and retrieve data.

BLAISE COMPUTING INC.

Circle **no. 98** on reader service card.

YOUR QUALITY CONNECTION...

Blaise Computing products are available from Programmer's Connection at these low discount prices with FREE UPS shipping for all US customers (express services are available at the shipping carrier's standard rate with no extra rush fees).

	List	Ours
C TOOLS PLUS	175	135
ASYNCH MANAGER Specify C or MS Pascal	175	135
C TOOLS	125	99
C TOOLS 2	100	79
PASCAL TOOLS for MS Pascal	125	99
PASCAL TOOLS 2 for MS Pascal	100	79
PASCAL TOOLS & PASCAL TOOLS, 2	175	135
TURBO ASYNCH PLUS for Turbo Pascal	100	83
TURBO POWER TOOLS PLUS Turbo Pascal	100	83
VIEW MANAGER Specify C or MS Pascal	275	199

Please refer to our main advertisement in this journal for more information about our services as well as the largest advertised selection of programmer's development tools specifically for IBM personal computers and compatibles.

PROGRAMMER'S QUALITY INTEGRITY Service SUPPORT PRICE CONNECTION

U.S.
800-336-1166
CANADA
800-225-1166
OHIO AND OVERSEAS
216-877-3781
CUSTOMER SERVICE 216-877-1110

programmer's connection

136 SUNNYSIDE ST. HARTVILLE, OHIO 44632

Listing Three *(Listing continued, text begins on page 40.)*

```
                callsys  OpenWindow
                lea      WindowPtr,A0
                move.l   D0,(A0)
;-------------- Set up the menus ------------------------------
                passtext ProjText1,Proj1     ;First must initialize all
                passtext ProjText2,Proj2     ;Intuition text structures.
                passtext ProjText3,Proj3
                passtext ProjText4,Proj4
                passtext ProjText5,Proj5
                passtext ProjText6,Proj6
                passtext ProjText7,Proj7

                lea      ProjItem2,A3        ;Then must include the text
                passitem ProjItem1,A3,ProjText1,#'N',#0
                lea      ProjItem3,A3        ;in menu item structures.
                passitem ProjItem2,A3,ProjText2,#'O',#9
                lea      ProjItem4,A3
                passitem ProjItem3,A3,ProjText3,#'S',#18
                lea      ProjItem5,A3
                passitem ProjItem4,A3,ProjText4,#'A',#27
                lea      ProjItem6,A3
                passitem ProjItem5,A3,ProjText5,#'P',#36
                lea      ProjItem7,A3
                passitem ProjItem6,A3,ProjText6,#'R',#45
                passitem ProjItem7,#0,ProjText7,#'Q',#54

                lea      ProjMenu,A0         ;Finally, must initialize the
                lea      EditMenu,A1         ;menu structure itself.
                move.l   A1,mu_NextMenu(A0)  ;pointer to next menu in list
                move     #0,mu_LeftEdge(A0)  ;place for title in menu strip
                move     #0,mu_TopEdge(A0)   ;ignored
                move     #0,mu_Height(A0)    ;ignored
                move     #100,mu_Width(A0)
                move     #MENUENABLED,mu_Flags(A0) ;menu is enabled
                lea      ProjName,A1
                move.l   A1,mu_MenuName(A0)
                lea      ProjItem,A1
                move.l   A1,mu_FirstItem(A0) ;head of menu item list

                passtext EditText1,Edit1     ;Now, repeat process for
                passtext EditText2,Edit2     ;the second menu.
                passtext EditText3,Edit3
                passtext EditText4,Edit4
                passtext EditText5,Edit5

                lea      EditItem2,A3
                passitem EditItem1,A3,EditText1,#'Z',#0
                lea      EditItem3,A3
                passitem EditItem2,A3,EditText2,#'X',#9
                lea      EditItem4,A3
                passitem EditItem3,A3,EditText3,#'C',#18
                lea      EditItem5,A3
                passitem EditItem4,A3,EditText4,#'V',#27
                passitem EditItem5,#0,EditText5,#'D',#36

                lea      EditMenu,A0
                move.l   #0,mu_NextMenu(A0)  ;end of the list
                move     #101,mu_LeftEdge(A0)
                move     #0,mu_TopEdge(A0)
                move     #75,mu_Width(A0)
                move     #0,mu_Height(A0)
                move     #MENUENABLED,mu_Flags(A0)
                lea      EditName,A1
                move.l   A1,mu_MenuName(A0)
                lea      EditItem1,A1
                move.l   A1,mu_FirstItem(A0)

                move.l   IntBase,A6
                move.l   WindowPtr,A0        ;window in question
                lea      ProjMenu,A1         ;first menu in strip
                callsys  SetMenuStrip        ;attach menu strip to window

;-------------- Initialize message ports for console ------------------
                lea      WritePort,A3        ;storage for pointer to write port
                move.l   #0,A5               ;unnamed ports - first in list
                jsr      CreatePort          ;initialize the port

                move.l   WritePort,A3        ;write port pointer
                lea      WriteMsg,A5         ;storage for pointer to IO block
                jsr      CreateStdIO         ;initialize IO block

                lea      ReadPort,A3         ;Repeat for read port.
                lea      ReadName,A5         ;has name - not first in list
                jsr      CreatePort

                move.l   ReadPort,A3
                lea      ReadMsg,A5
                jsr      CreateStdIO

;-------------- Open and attach the console device --------------------
                move.l   WriteMsg,A3         ;output IO request block
                move.l   ReadMsg,A5          ;input IO request block
```

```
        move.l    WindowPtr,A4          ;window for this console
        jsr       OpenConsole
        cmp       #0,D0
        beq       GoOn
        rts                             ;unsuccessful opening ends program

;------------- Identify signal bits -----------------------------------
GoOn    move.l    WindowPtr,A0
        move.l    wd_UserPort(A0),A0    ;message port for Intuition
        move.b    MP_SIGBIT(A0),D0      ;Intuition signal bit
        lea       IntSigBit,A0
        move.b    D0,(A0)               ;save it
        move.l    ReadPort,A0
        move.b    MP_SIGBIT(A0),D0      ;console signal bit
        lea       ConSigBit,A0
        move.b    D0,(A0)               ;save it

;-------------- Queue up an initial read request ---------------------
        move.l    ReadMsg,A1            ;console IO request block
        lea       letter,A4             ;place to put character read
        jsr       QueueRead             ;queue up a message

;------------ Wait for Intuition or console event --------------------
Event   clr.l     D1
        move.b    IntSigBit,D1
        clr.l     D0
        bset.l    D1,D0                 ;sys will look for Intuition event
        clr.l     D1
        move.b    ConSigBit,D1
        bset.l    D1,D0                 ;system looks for console evt, too
        move.l    _AbsExecBase,A6
        callsys   Wait                  ;wait for event to occur

        clr.l     D1                    ;Note - now a bit in D0 is set
        move.b    IntSigBit,D2          ;to correspond to signal causing
        bset.l    D2,D1                 ;event.
        cmp.l     D1,D0                 ;Intuition event?
        beq       IntuitionEvent

        clr.l     D1
        move.b    ConSigBit,D2
        bset.l    D2,D1
        cmp.l     D1,D0                 ;console event?
        beq       ConsoleEvent

        bra       Event                 ;fail-safe trap-should never get here

;------------- Handle Intuition events -------------------------------
IntuitionEvent
        move.l    WindowPtr,A0
        move.l    wd_UserPort(A0),A0    ;Intuition's message port
        move.l    _AbsExecBase,A6
        callsys   GetMsg                ;retrieve the input message
        beq       Event                 ;no message present

        move.l    D0,A1                 ;GetMsg returns address of message in D0
        move.l    im_Class(A1),D0       ;type of event
        cmp       #CLOSEWINDOW,D0       ;was window close box clicked?
        beq       CloseAndQuit

        cmp       #MENUPICK,D0          ;menu choice made?
        beq       MenuEvent

DoneWithEvent
        move.l    _AbsExecBase,A6
        callsys   ReplyMsg              ;remove message so it can be reused
        bra       Event

MenuEvent
        move      im_Code(A1),D0        ;menu & menu item number
        beq       DoneWithEvent         ;user backed out before chosing

        move      D0,D1                 ;save the code
        and       #%0000000000011111,D0 ;get menu number
        cmp       #0,D0                 ;project menu?
        bne       DoneWithEvent         ;Project menu is the only one
                                        ;trapped by this program!!!!

        lsr       #5,D1
        and       #%0000000000111111,D0 ;get menu item number
        cmp       #6,D1                 ;Quit selected?
        bne       DoneWithEvent         ;Quit is the only option
                                        ;implemented by this program!!!!!

CloseAndQuit
        move.l    ReadMsg,A1
        ABORTIO                         ;remove last event from queue

        move.l    ReadMsg,A1
        move.l    _AbsExecBase,A6
        callsys   CloseDevice           ;close the console

        move.l    WindowPtr,A0
        move.l    IntBase,A6
        callsys   CloseWindow           ;close the window
```

(continued on next page)

MAC AND AMIGA

Listing Three *(Listing continued, text begins on page 40.)*

```
                 move.l    ScreenPtr,A0
                 move.l    IntBase,A6
                 callsys   CloseScreen         ;close the custom screen

                 rts                           ;return to DOS

;----------- Handle console events ----------------------------------
        ConsoleEvent
                 move.l    ReadMsg,A0
                 move.l    _AbsExecBase,A6
                 callsys   GetMsg              ;retrieve the message
                 move.l    WriteMsg,A1         ;output IO request block
                 lea       letter,A4           ;place where character is stored
                 jsr       ConPutChar          ;display the character
                 move.b    letter,D0
                 cmp       #$D,D0              ;was character a <cr>?
                 bne       MoreLetters
                 move.b    #$A,(A4)            ;put line feed in letter
                 move.l    WriteMsg,A1
                 jsr       ConPutChar          ;add a line feed to the <cr>

        MoreLetters
                 move.l    ReadMsg,A1
                 move.l    DevAdd,IO_DEVICE(A1)
                 jsr       QueueRead ;go get another letter

                 bra       Event

;--------- Subroutine to load Intuition text structures ---------------
        SetText  move.b    #FrontPen,it_FrontPen(A0)  ;colors for drawing
                 move.b    #BackPen,it_BackPen(A0)
                 move.b    #0,it_DrawMode(A0)
                 move      #2,it_LeftEdge(A0)  ;posn rel to container
                 move      #1,it_TopEdge(A0)
                 move.l    #0,it_ITextFont(A0) ;use default font
                 move.l    A1,it_IText(A0)     ;pointer to the text structure
                 move.l    #0,it_NextText(A0)  ;no link to other txt structs
                 rts

;---------- Subroutine to load menu item structures ------------------
        SetItem  move.l    A1,mi_NextItem(A0)  ;pointer to next item in list
                 move      #2,mi_LeftEdge(A0)  ;posn rel to container
                 move      D1,mi_TopEdge(A0)
                 move      #100,mi_Width(A0)
                 move      #9,mi_Height(A0)
                 move      #ITEMTEXT+COMMSEQ+ITEMENABLED+HIGHCOMP,mi_Flags(A0)
                 move.l    #0,mi_MutualExclude(A0)      ;no mutually exclusive items
                 move.l    A2,mi_ItemFill(A0)  ;pointer to text structure
                 move.b    D0,mi_Command(A0)   ;keyboard equivalent
                 move.l    #0,mi_SubItem(A0)   ;no subitems
                 move      #0,mi_NextSelect(A0) ;no associated items
                 rts

;------------ Create a message port ----------------------------------
;NOTE - this subroutine is an assembly language version of the C source
;code provided in the Amiga ROM kernel manual. It needs error trapping
;after the system calls to be complete.
;Load address of pointer to message port structure in A3.
;Load pointer to name for message port in A5.
```

(continued on page 79)

4 REASONS TO CHOOSE
PROGRAMMER'S CONNECTION:

QUALITY

SUPPORT

PRICE &

INTEGRITY

We carry the finest selection of the best programmer's development tools specifically for IBM Personal Computers and compatibles. They are the latest versions and most come with 30-day documentation evaluation periods or 30-day return guarantees.

We firmly believe that high quality must be present throughout every aspect of our service. So to make sure that we maintain such high standards, we include a service questionnaire with every purchase. We're very interested in what our customers have to say.

Our courteous, knowledgeable, non-commissioned salespeople are always ready to assist you. We also have experienced technical consultants on staff who can answer questions about products and provide sound, unbiased advice. We'll support you before and after you make your purchase. Your satisfaction is very important to us.

Our buying power enables us to offer you the lowest prices without sacrificing service. UPS shipping is FREE to all U.S. customers. There are no extra charges for credit cards, CODs, purchase orders or special handling (except for export preparation).

Quite simply, the discount prices listed with the products on the next two pages are all you pay. There are no hidden add-on charges.

When we started Programmer's Connection in 1984, we dedicated ourselves to providing high quality personal service to every customer. Since then, we've quickly grown to be the leading independent dealer in this industry.

We're very proud of the trust we've earned from our customers and we pledge always to be worthy of it.

As we enter 1987, we'd like to extend a warm thank you to our customers and wish everyone a Happy New Year!

If you've bought from us before, we look forward to serving you again. And if you're not yet familiar with our one-stop service, we invite you to give us a call.

It's our commitment to quality, support, low prices and integrity that makes us your best source for the programming tools you need. So make the connection today and discover the value and convenience of our one-stop service for yourself. You'll be glad you did!

Call Toll Free
United States 800-336-1166
Canada 800-225-1166
Ohio & Overseas 216-877-3781

PROGRAMMER'S CONNECTION
QUALITY INTEGRITY SUPPORT PRICE
Service

Turn the page for our product listing and ordering information.

apl language

APL*PLUS/PC by STSC	595	429
APL*PLUS/PC Spreadsheet Mgr by STSC	195	139
APL*PLUS/PC Tools Vol 1 by STSC	295	199
APL*PLUS/PC Tools Vol 2 by STSC	85	59
APL*PLUS/UNX For AT XENIX by STSC	995	695
Btrieve ISAM File Mgr by SoftCraft	245	194
Financial/Statistical Library by STSC	275	195
Pocket APL by STSC	95	69
STATGRAPHICS by STSC	795	579

artificial intelligence

1st-CLASS by Programs in Motion	495	399
APT from Solution Systems	65	CALL
Arity Products Various	CALL	CALL
AutoIntelligence by IntelligenceWare	990	CALL
ESP ADVISOR by Expert Systems Intl.	895	839
PROLOG-2 Interface	395	369
ExpertEDGE Advanced by Human Edge	2500	CALL
ExpertEDGE Professional by Human Edge	5000	CALL
Experteach II by IntelligenceWare	475	359
EXSYS Development Software by EXSYS	395	319
GCLISP Golden Common LISP by Gold Hill	495	CALL
GCLISP 286 Developer by Gold Hill	1190	CALL
Insight 1 by Level Five Research	95	75
Insight 2+ by Level Five Research	485	379
Intelligence/Compiler IntelligenceWare	990	749
Logic-Line Series 1 by Thunderstone	90	85
Logic-Line Series 2 by Thunderstone	125	115
Logic-Line Series 3 by Thunderstone	150	139
LPA microPROLOG by Prog Logic Systems	99	89
with APES	149	129
LPA Professional microPROLOG	395	339
with APES	650	569
Microsoft LISP Common LISP	250	169
PC Scheme by Texas Instruments	95	85
Personal Consultant Easy by TI	495	439
Personal Consultant Plus by TI	2950	2599
Personal Consultant Runtime	CALL	CALL
PROLOG-2 Interpreter by ESI	450	419
PROLOG-2 Interpreter and Compiler	895	839
QNIAL by NIAL Systems	375	349
TransLISP from Solution Systems	95	CALL
Turbo PROLOG Compiler by Borland Intl.	100	75

assembly language

386 ASM/LINK Cross Asm by Phar Lap	495	CALL
8088 Assembler w Z-80 Trans by 2500 AD	100	89
ASMLIB Function Library by BC Assoc	149	129
asmTREE B-Tree Dev System by BC Assoc	395	339
Cross Assemblers Various 2500 AD	CALL	CALL
Microsoft Macro Assembler	150	98
Norton Utilities by Peter Norton	100	59
Turbo EDITASM by Speedware	99	84
UniWare Cross Assemblers Various. New	295	CALL
Visible Computer: 8088 Software Masters	80	65

basic language

BetterBASIC by Summit Software	200	129
8087 Math Support	99	75
Btrieve Interface	99	75
C Interface	99	75
Run-time Module	250	169
EXIM Services Toolkit by EXIM. New	CALL	CALL
Finally by Komputerwerks. New	99	85
Inside Track from Micro Help	65	55
MACH 2 by Micro Help	75	65
Microsoft QuickBASIC	99	65
Peeks 'n Pokes from MicroHelp	45	39
Professional BASIC by Morgan	99	75
8087 Math Support	50	42
Stay-Res by MicroHelp. New	95	85
True Basic w BASICA Converter. New Version	200	99
True Basic w Converter & Run-time	295	199
Advanced String Library	50	45
Asynch Communication Support	50	45
BASICA Converter	50	45
Btrieve Interface	50	45
Developer's Toolkit	50	45
Formlib	50	45
Hercules Graphic Support	50	45
Run-time Module	150	109
Sorting & Searching	50	45

blaise products

ASYNCH MANAGER Specify C or Pascal	175	135
C TOOLS PLUS	175	135
EXEC Program Chainer	95	75
PASCAL TOOLS	125	99
PASCAL TOOLS 2	100	79
PASCAL TOOLS & PASCAL TOOLS 2	175	135
RUNOFF Text Formatter	50	45
TURBO ASYNCH PLUS	100	83
TURBO POWER TOOLS PLUS	100	83
VIEW MANAGER Specify C or Pascal	275	199

borland products

REFLEX Data Base System	150	99
REFLEX Workshop	70	48
REFLEX & REFLEX Workshop	200	129
Turbo DATABASE TOOLBOX	70	48
Turbo EDITOR TOOLBOX	70	48
Turbo GAMEWORKS TOOLBOX	70	48
Turbo GRAPHIX TOOLBOX	70	48
Turbo LIGHTNING	100	65
Turbo PASCAL with 8087 and BCD	100	65
Turbo Prolog Compiler	100	65
Turbo TUTOR for Turbo PASCAL	40	28
Word Wizard	70	48
Word Wizard and Turbo Lightning	150	99

c++

C++ from Guidelines. New Version	195	179

c compilers

C86PLUS by Computer Innovations. New	497	CALL
Datalight C Compiler Small Model	60	49
Datalight Developer Kit w Large Model	99	79
DeSmet C w Debugger	159	145
DeSmet C w Debugger & Large Case	209	193
Eco-C Development System by Ecosoft	125	89
Lattice C Compiler from Lattice	500	275
Mark Williams Let's C	75	58
with csd Source Debugger	150	109
Mark Williams MWC-86	495	289
Microsoft C with CodeView	450	275
UniWare 68000/10/20 Cross Compiler. New	595	CALL
Wizard C Combo by Wizard Systems	750	599
Wizard C Compiler	450	359
ROM Development Pkg	350	299

c interpreters

C-terp by Gimpel, Specify compiler	300	235
C Trainer with Book by Catalytix	122	CALL
Instant C by Rational Systems	500	CALL
Introducing C by Computer Innovations	125	104
Run/C from Lifeboat	150	89
Run/C Professional from Lifeboat	250	169

c utilities

See also Blaise, GSS, Lattice, Microsoft, Phoenix, Polytron, SoftCraft and XENIX sections.

APT by Shaw American Technology	395	299
Basic C Library by C Source	175	129
C Essentials by Essential Software	100	CALL
C-ISAM by Informix	225	195
C to dBase by Computer Innovations	150	135
c-tree & r-tree Combo Package. New	650	529
c-tree ISAM File Manager by FairCom	395	329
r-tree Report Generator	295	249
C Utility Library by Essential Software	185	135
C Windows by Syscom	100	89
C Wings by Syscom	50	45
CI ROMPac by Computer Innovations	195	CALL
dbQUERY All Varieties by Raima	CALL	CALL
dbVISTA Single-User DBMS by Raima	195	155
with Source Code	495	425
dbVISTA Multi-User DBMS by Raima	495	425
with Source Code	990	845
dBx dBase C Translator by Desktop AI	350	325
with Library Source Code	550	499
Entelekon Combo Package	200	169
C Function Library	130	109
C Windows	130	109
Superfonts for C	50	43
Essential Comm Library		
with Debugger. New	250	195
Breakout Debugger Any language. New	125	99
Essential Comm Library. New	185	135
Essential Graphics by Essential Software	250	195
Flash-up Windows by Software Bottling	90	79
GraphiC Mono v2.2 by Sci Endeavors	280	209
GraphiC Color v3.0 by Sci Endeavors	350	289
GRAFLIB by The Librarian	175	CALL
Greenleaf Comm Library by Greenleaf	185	134
Greenleaf Data Windows by Greenleaf	225	189
with Source Code	450	379
Greenleaf Functions by Greenleaf	185	134
The HAMMER by OES Systems	195	149
HALO by Media Cybernetics	300	209
HELP/Control by MDS	125	109
MetaWINDOWS No Royalties	185	115
MetaFONTS	80	58
MetaWINDOWS/Plus by Metagraphics	235	189
MetaFONTS/Plus	235	189
On-line Help from Opt-Tech Data Proc	149	109
PANEL by Roundhill Computer Systems	295	224
PC Lint by Gimpel Software	139	105
PLOTHI by The Librarian	175	CALL
PLOTHP by The Librarian	175	CALL
Sci Subroutine Library by Peerless	175	138
Vector87 by Vectorplex Data Systems. New	150	135
Vitamin C by Creative Prog. New Version	225	CALL
VC Screen Forms Designer	100	84
Zview by Data Management Consultants	245	189

cobol language

Micro Focus COBOL Workbench	4000	3379
Micro Focus Level II COBOL	1500	549
COGRAPHICS	250	199
COMATH	200	159
FORMS-2	300	259
Level II Animator	900	349
Level II SOURCEWRITER	2000	CALL
Micro Focus Level II COBOL for Novell	2000	1699
Micro Focus Micro/SPF	175	149
Micro Focus Professional COBOL	3000	2295
Multi-user Runtime for PC Network	500	429
Microsoft COBOL Compiler	700	439
for XENIX	995	635
Realia COBOL	995	785
RM/COBOL by Ryan-McFarland	950	639
RM/COBOL 8X ANSI 85 by Ryan-McFarland	1250	895

debuggers & profilers

386 DEBUG Cross Debugger by Phar Lap	195	159
Advanced Trace-86 by Morgan Computing	175	125
CI Probe by Computer Innovations	225	189
Codesifter Profiler by David Smith	119	98
Codesmith-86 by Visual Age	145	108
DSD86 by Soft Advances	70	65
DSD87 by Soft Advances	100	89
Periscope I by The Periscope Company	295	245
Periscope II w NMI Breakout Switch	145	109
Periscope II-X Software only	115	84
The PROFILER with Source Code by DWB	125	94
The WATCHER Profiler by Stony Brook	60	55

forth language

CFORTH Native Code Compiler by LMI	300	239
Forth/83 Metacompiler Specify Target	750	599
PC/Forth by Laboratory Microsystems	150	119
PC/Forth+ by Laboratory Microsystems	250	209
Advanced Color Graphics Support	100	79
Enhanced Graphics Support	200	159
Intel 8087 Support	100	79
Interactive Symbolic Debugger	100	79
Native Code Optimizer	200	159
PCTERM Modem Pgm for Smartmodem	100	79
Software Floating Point	100	79
UR/Forth by Laboratory Microsystems. New	350	279
Object Module Libraries. New	500	395
Source Code License. New	1500	995

fortran language

50 MORE: FORTRAN by Peerless Engr	125	99
ACS Time Series Alpha Computer Service	495	419
Btrieve ISAM File Mgr by SoftCraft	245	194
Essential Graphics by Essential Software	250	195
For-Winds Alpha Computer Service	90	78
Forlib-Plus Alpha Computer Service	70	54
FORTLIB by The Librarian	95	CALL
FORTRAN Addenda by Impulse Engr	95	89
FORTRAN Addendum by Impulse Engr	165	149
GRAFLIB by The Librarian	175	CALL
HALO by Media Cybernetics	300	209
I/O PRO w No Limit Library by MEF	390	349
Microcompatibles Combo Package	240	219
Grafmatic	135	119
Plotmatic	135	119
Microsoft FORTRAN Compiler	350	204
No Limit by MEF Environmental	129	115
PANEL Screen Designer by Roundhill	295	224
PLOTHI by The Librarian	175	CALL
PLOTHP by The Librarian	175	CALL
RM/FORTRAN Ryan-McFarland	595	389
Sci Subroutine Library by Peerless	175	138
Statistician Alpha Computer Service	295	249
Strings & Things Alpha Computer Service	70	54
Vector87 by Vectorplex Data Systems	150	135

gss products

GSS Graphics Development Toolkit	495	375
GSS Kernel System for DOS	495	375
GSS Kernel System for IBM RT	795	649
GSS Metafile Interpreter	295	239
GSS Plotting System	495	375

lattice products

Lattice C Compiler from Lattice	500	275
with Library Source Code	900	495
C Cross Reference Generator	50	38
with Source Code	200	145
C-Food Smorgasbord Function Library	150	95
with Source Code	300	185
C-Sprite Source Level Debugger	175	129
Curses Screen Manager	125	89
with Source Code	250	178
dBC dBase File Manager for C	250	178
with Source Code	500	356
LMK Make Facility	195	139
RPG II Compiler No Royalties	750	635
RPG II Combo with SEU & Sort/Merge. New	1100	939
SecretDisk File Encryption Utility	120	89
SideTalk Resident Communications	120	89
Text Management Utilities	120	89
TopView Toolbasket Function Library	250	178
with Source Code	500	356
Z-80 C Cross Compiler	500	356

The Peak of Performance

SCALE THE HEIGHTS OF PRODUCTIVITY

Sure, you've proven that in your hands a computer is a productive tool. But if you haven't teamed up with a SemiDisk you have heights yet to climb!

IT'S NO MERE RAMDISK

SemiDisk has been leading the way for Disk Emulators since their inception. If you've seen RAMdisks you know what it's like to load programs in an instant, and read or write files without delay. Unlike alternatives, the SemiDisk offers up to 8 megabytes of instant-access storage while leaving your computer's main memory free for what it does best - computing!

KEEP A GRIP ON DATA

Go ahead, turn off your computer. Take a vacation. With the battery backup option, your valuable data will be there in the morning even if you aren't. You'll sleep better knowing not even a 5 hour blackout will sabotage your files.

NEW LOWER SEMIDISK PRICES THAT WON'T SNOW YOU UNDER

	512K	2Mbyte
IBM PC, XT, AT	$495	$995
Epson QX-10	$595	$995
S-100, SemiDisk II	$799	$1295
S-100, SemiDisk I	$595	——
TRS-80 II, 12, 16	$695	$1295
Battery Backup Unit	$130	$130

Software drivers available for CP/M 80, MS-DOS, ZDOS, TurboDOS, and VALDOCS 2.

SEMIDISK

SemiDisk Systems, Inc.
P.O. Box GG, Beaverton, Oregon 97075
503-626-3104

Call 503-646-5510 for CBBS/NW, and 503-649-8327 for CBBS/Aloha, all SemiDisk equipped computer bulletin boards, (300/1200/2400 baud) SemiDisk, SemiSpool trademarks of SemiDisk Systems.

Circle **no. 85** on reader service card.

Listing Three *(Listing continued, text begins on page 40.)*

```
CreatePort
                move      #-1,D0               ;no preference for signal bit
                move.l    _AbsExecBase,A6
                callsys   AllocSignal          ;allocate a signal bit for port

                move.b    D0,D7                ;save signal bit
                clr.l     D1
                bset.l    #16,D1               ;(clear)
                bset.l    #0,D1                ;(public)  requirements
                move      #MP_SIZE,D0          ;number of bytes needed
                move.l    _AbsExecBase,A6
                callsys   AllocMem             ;memory for message port structure

                move.l    D0,(A3)              ;save pointer to message port
                move.l    D0,A4
                move.l    #0,D0
                move.l    _AbsExecBase,A6
                callsys   FindTask  ;initialize task control block

                move.l    A5,LN_NAME(A4)       ;port's name
                move.b    #0,LN_PRI(A4)        ;port's priority
                move.b    #NT_MSGPORT,LN_TYPE(A0)  ;type of port
                move.b    #PA_SIGNAL,MP_FLAGS(A0)
                move.b    D7,MP_SIGBIT(A4)     ;signal bit
                move.l    D0,MP_SIGTASK(A4)    ;address of task ctrl block

                cmp.l     #0,A5                ;is name specified?
                beq       Port2                ;head of list of ports
                move.l    A4,A1
                move.l    _AbsExecBase,A6
                callsys   AddPort              ;add this port to list
                rts

        Port2   lea       MP_MSGLIST(A4),A0
                NEWLIST   A0                   ;initialize a new list of ports
                rts

;--------------- Create a standard IO request structure ---------------
;NOTE - this subroutine is an assembly language version of the C source
;code provided in the Amiga ROM kernel manual. It needs error trapping
;after the system calls to be complete.
;Load pointer to message port in A3.
;Load address of pointer for standard IO structure in A5.

CreateStdIO
                clr.l     D1
                bset.l    #16,D1
                bset.l    #0,D1
                move.l    #IOSTD_SIZE,D0
                move.l    _AbsExecBase,A6
                callsys   AllocMem             ;space for IO request block

                move.l    D0,(A5)              ;save the pointer
                move.l    D0,A0
                move.b    #NT_MESSAGE,LN_TYPE(A0)  ;type of structure
                move.b    #0,LN_PRI(A0)        ;priority
                move.l    A3,MN_REPLYPORT(A0)  ;address of message port
                rts

;-------- Subroutine to open the console device --------------------
;NOTE - this subroutine is an assembly language version of the C source
;code provided in the Amiga ROM kernel manual.
;Load pointer to WriteMsg in A3.
;Load pointer to ReadMsg in A5.
;Load pointer to Window in A4.

        OpenConsole
                move.l    A4,IO_DATA(A3)       ;pointer to window record
                move      #nw_SIZE,IO_LENGTH(A3)        ;size of window record
                lea       ConDev,A0 ;name of device
                move.l    #0,D0
                move.l    A3,A1
                move.l    #0,D1
                move.l    _AbsExecBase,A6
                callsys   OpenDevice

                move.l    IO_DEVICE(A3),IO_DEVICE(A5) ;save device pointers
                move.l    IO_DEVICE(A3),DevAdd
                move.l    IO_UNIT(A3),IO_UNIT(A5)
                rts

;------ Subroutine to queue up a read request to the console -----------
;NOTE - this subroutine is an assembly language version of the C source
;code provided in the Amiga ROM kernel manual.
;Load pointer to read message in A1.
;Load pointer to storage space for character in A4.

        QueueRead
                move      #CMD_READ,IO_COMMAND(A1) ;type of operation
                move.l    A4,IO_DATA(A1)       ;where data should be placed
                move.l    #1,IO_LENGTH(A1)     ;number of bytes to read
                move.l    _AbsExecBase,A6
                callsys   SendIO
                rts
```

(continued on next page)

Listing Three

(Listing continued, text begins on page 40.)


```
;-------- Subroutine to print a single character in a console window ----
;NOTE - this subroutine is an assembly language version of the C source
;code provided in the Amiga ROM kernel manual.
;Load pointer to write message in A1
;Load pointer to character to be printed in A4.

        ConPutChar
                move      #CMD_WRITE,IO_COMMAND(A1) ;type of operation
                move.l    A4,IO_DATA(A1)      ;where data will come from
                move.l    #1,IO_LENGTH(A1)    ;number of bytes to output
                move.l    _AbsExecBase,A6
                callsys   DoIO
                rts

**************************** Data Structures ******************************
        DevAdd              ds.l      1
        TheScreen ds.b      ns_SIZEOF
        TheWindow ds.b      nw_SIZE
        IntBase             ds.l      1
        IntName             dc.b      'intuition.library',0
        WindowPtr ds.l      1
        WindowTitle         dc.b      'Text Window',0
        ScreenPtr ds.l      1
        ScreenTitle         dc.b      'Dr. Dobbs Journal',0
        ConDev              dc.b      'console.device',0,0
        ;NOTE - extra 0 included to keep addresses even

        ProjMenu  ds.b      mu_SIZEOF
        ProjName  dc.b      'Project',0
        Proj1               dc.b      'New',0
        ProjText1 ds.b      it_SIZEOF
        ProjItem1 ds.b      mi_SIZEOF

        Proj2               dc.b      'Open',0,0
        ProjText2 ds.b      it_SIZEOF
        ProjItem2 ds.b      mi_SIZEOF

        Proj3               dc.b      'Save',0,0
        ProjText3 ds.b      it_SIZEOF
        ProjItem3 ds.b      mi_SIZEOF

        Proj4               dc.b      'Save As',0
        ProjText4 ds.b      it_SIZEOF
        ProjItem4 ds.b      mi_SIZEOF

        Proj5               dc.b      'Print',0
        ProjText5 ds.b      it_SIZEOF
        ProjItem5 ds.b      mi_SIZEOF

        Proj6               dc.b      'Print As',0,0
        ProjText6 ds.b      it_SIZEOF
        ProjItem6 ds.b      mi_SIZEOF

        Proj7               dc.b      'Quit',0,0
        ProjText7 ds.b      it_SIZEOF
        ProjItem7 ds.b      mi_SIZEOF

        EditMenu  ds.b      mu_SIZEOF
        EditName  dc.b      'Edit',0,0
        Edit1               dc.b      'Undo',0,0
        EditText1 ds.b      it_SIZEOF
        EditItem1 ds.b      mi_SIZEOF

        Edit2               dc.b      'Cut',0
        EditText2 ds.b      it_SIZEOF
        EditItem2 ds.b      mi_SIZEOF

        Edit3               dc.b      'Copy',0,0
        EditText3 ds.b      it_SIZEOF
        EditItem3 ds.b      mi_SIZEOF

        Edit4               dc.b      'Paste',0
        EditText4 ds.b      it_SIZEOF
        EditItem4 ds.b      mi_SIZEOF

        Edit5               dc.b      'Erase',0
        EditText5 ds.b      it_SIZEOF
        EditItem5 ds.b      mi_SIZEOF

        ReadPort  ds.l      1
        ReadMsg             ds.l      1
        ReadName  dc.b      'Read',0,0
        WritePort ds.l      1
        WriteMsg  ds.l      1
        IntSigBit ds.b      1
        ConSigBit ds.b      1
        letter              ds.b      1
```

End Listings

Listing One *(Text in December)*

```
/* A32000.C - Series 32000 assembler
   850903 rr fix addr ext, scaled index, acp, cxp 0.10
   850902 rr add scaled index logic 0.09
   850828 rr fix enter, setcfg, lpr/spr, index 0.08
   850809 rr add equate logic 0.07
   850730 rr add binary search in lookup 0.06
   850729 rr symbol table mods, reglist 0.05

Still need:
   --- register names for lpr/spr
   --- linkable modules

Note: While 68000 is hilo (high-bytes at lower memory
   addresses), 32000 is lohi (low-bytes at lower memory
   addresses, like the Z80).

   This is a 3-pass assembler; 3 passes to make
   sure that relative branches are computed correctly. */

#define EOF -1

#define SYMSIZ 1024        /* symbol table size */

char inpbuf[ 256 ];        /* input buffer */
int inpcnt, inpptr;        /* input counter, pointer */

char word_buffer[ 128 ];       /* buff for current word */
char ambig_buffer[ 128 ];      /* ambiguous refs here */

char listline[ 81 ];       /* line of listing output */
int listop, listcp;        /* pointers for list output */

int paren = 0;             /* used in gchar() */
int brack = 0;
int quote = 0;

int iwparen = 0;           /* used in inword() */

int errors = 0;            /* count of errors */

char *word;                /* pointer to current word */
char *ambig[ 10 ];         /* filled in by match */
int ambcnt = 0;            /* count of pointers in ambig[] */

int pass;                  /* pass = 1, 2 or 3 */
long int asmadr, codadr; /* assembly addr, code addr */
char filename[ 30 ];

int fasm, fobj;            /* file numbers */

char objbuf[ 64 ];         /* object byte buffer */
long int objadr;           /* addr of first byte of buf */
int objcnt = 0;            /* count of bytes in buffer */

struct {                   /* Symbol table */
   char *snam;             /* symbol name */
   long int sval;          /* value */
} symbol[ SYMSIZ ];

int symcnt;                /* count of symbols */

char hexchr[ 17 ] = "0123456789abcdef";

/* --- 32000 opcodes --- */

/* Note: Shortest form of opcode must be listed first. */

#define MAXOP 149

/* the opcode binary value should be a string of bits,
   e.g. 0111xxxxx000b the opcode opopt character is used
   to specify special operands, etc. */

/* opopts used here for the 32000 are:
   blank      nothing special
   a    gen
   b    gen short
   c    gen gen
   d    00000 short
   e    gen gen reg
   f    reglist save/enter
```

```
   g     reglist restore/exit
   h     00000 gen (sfsr)
   i     inss/exts
   j     movs/skps/cmps
   k     setcfg
   l     procreg, gen for lpr/spr
   m     index (operand order)
   n     ret/rett - postbyte
   o     movm
   p     cxp (disp after instruction) */

struct {
   char *onam;  /* opcode name */
   int ocnt;    /* operand count, negative if PC-rel */
   char *obin;  /* opcode binary value */
   char oopt;   /* opcode opopt char */
} opcode[ MAXOP ] = {

/* Format 1 ops (16) */

   "bsr",       -1,  "02h",    ' ',
   "ret",       1,   "12h",    'n',
   "cxp",       1,   "22h",    'p',
   "rxp",       1,   "32h",    'n',
   "rett",      1,   "42h",    'n',
   "reti",      0,   "52h",    ' ',
   "save",      1,   "62h",    'f',
   "restore",   1,   "72h",    'g',
   "enter",     2,   "82h",    'f',
   "exit",      1,   "92h",    'g',
   "nop",       0,   "0a2h",   ' ',
   "wait",      0,   "0b2h",   ' ',
   "dia",       0,   "0c2h",   ' ',
   "flag",      0,   "0d2h",   ' ',
   "svc",       0,   "0e2h",   ' ',
   "bpt",       0,   "0f2h",   ' ',

/* Conditional branches (15) */

   "beq",       -1,   "0ah",    'b',
   "bne",       -1,   "1ah",    'b',
   "bcs",       -1,   "2ah",    'b',
   "bcc",       -1,   "3ah",    'b',
   "bhi",       -1,   "4ah",    'b',
   "bls",       -1,   "5ah",    'b',
   "bgt",       -1,   "6ah",    'b',
   "ble",       -1,   "7ah",    'b',
   "bfs",       -1,   "8ah",    'b',
   "bfc",       -1,   "9ah",    'b',
   "blo",       -1,   "0aah",   'b',
   "bhs",       -1,   "0bah",   'b',
   "blt",       -1,   "0cah",   'b',
   "bge",       -1,   "0dah",   'b',
   "br",        -1,   "0eah",   'b',

/* Format 2 ops (7) */

   "addq?",     2,    "xxxxxxxxx00011iib",    'e',
   "cmpq?",     2,    "xxxxxxxxx00111iib",    'e',
   "spr?",      2,    "xxxxxxxxx01011iib",    'l',
   "lpr?",      2,    "xxxxxxxxx11011iib",    'l',

   "seq?",      1,    "xxxxx000001111iib",    'a',
   "sne?",      1,    "xxxxx000101111iib",    'a',
   "scs?",      1,    "xxxxx001001111iib",    'a',
   "scc?",      1,    "xxxxx001101111iib",    'a',
   "shi?",      1,    "xxxxx010001111iib",    'a',
   "sls?",      1,    "xxxxx010101111iib",    'a',
   "sgt?",      1,    "xxxxx011001111iib",    'a',
   "sle?",      1,    "xxxxx011101111iib",    'a',
   "sfs?",      1,    "xxxxx100001111iib",    'a',
   "sfc?",      1,    "xxxxx100101111iib",    'a',
   "slo?",      1,    "xxxxx101001111iib",    'a',
   "shs?",      1,    "xxxxx101101111iib",    'a',
   "slt?",      1,    "xxxxx110001111iib",    'a',
   "sge?",      1,    "xxxxx110101111iib",    'a',
   "st?",       1,    "xxxxx111001111iib",    'a',
   "sf?",       1,    "xxxxx111101111iib",    'a',

/* The acb instruction 3rd operand is a relative jump */
```

(continued on page 85)

Clipper is the fastest dBase III and dBase III Plus™ compiler available. Nothing else comes close. When performance counts, experts rely on Clipper for more speed, more power, and more creative freedom. You can, too. Call for details.
• Clipper compiled programs run 2 to 20 times faster.
• No royalties...no runtime fees.
• Source code security.
• User defined functions.
• Arrays.
• Simple menu commands.
• Context sensitive help can be included with programs.
• More fields; more memory variables.
• Call C and Assembly programs.
• Complete debugging facilities.
• Multiple file relationships.
• IBM PC, XT, AT, 3270 compatible™.
• Multi-user capability.
Clipper™
CLIPPER. THE dBASE COMPILER.
A WINNING PERFORMANCE EVERY TIME.
Nantucket™
Nantucket Corporation
5995 South Sepulveda Boulevard
Culver City, California 90230
(213) 390-7923
Outside California call toll-free:
1-800-251-8438
dBase, dBase III, and dBase III Plus are trademarks of Ashton-Tate, Inc.
IBM PC, XT, AT, and 3270 are trademarks of International Business Machines Corporation.
Clipper and Nantucket are trademarks of Nantucket Corporation
LOOK FOR CLIPPER™
Autumn '86
IT MAKES NETWORKING EASY.
Circle no. 220 on reader service card.

Listing One *(Listing continued)*

```
   "acb?",      -3,   "xxxxxxxxx10011iib",   'e',
   "movq?",      2,   "xxxxxxxxx10111iib",   'e',

/* Format 3 instructions (7) */

   "cxpd",       1,   "xxxxx00001111111b",   'a',
   "bicpsr?",    1,   "xxxxx001011111iib",   'a',
   "jump",       1,   "xxxxx01001111111b",   'a',
   "bispsr?",    1,   "xxxxx011011111iib",   'a',
   "adjsp?",     1,   "xxxxx101011111iib",   'a',
   "jsr",        1,   "xxxxx11001111111b",   'a',
   "case?",      1,   "xxxxx111011111iib",   'a',

/* Format 11 ops (16) -
   moved here so wildcards won't interfere */

   "addf",       2,   "xxxxxxxxxx00000110111110b",   'c',
   "addl",       2,   "xxxxxxxxxx00000010111110b",   'c',
   "movf",       2,   "xxxxxxxxxx00010110111110b",   'c',
   "movl",       2,   "xxxxxxxxxx00010010111110b",   'c',
   "cmpf",       2,   "xxxxxxxxxx00100110111110b",   'c',
   "cmpl",       2,   "xxxxxxxxxx00100010111110b",   'c',
   "subf",       2,   "xxxxxxxxxx01000110111110b",   'c',
   "subl",       2,   "xxxxxxxxxx01000010111110b",   'c',
   "negf",       2,   "xxxxxxxxxx01010110111110b",   'c',
   "negl",       2,   "xxxxxxxxxx01010010111110b",   'c',
   "divf",       2,   "xxxxxxxxxx10000110111110b",   'c',
   "divl",       2,   "xxxxxxxxxx10000010111110b",   'c',
   "mulf",       2,   "xxxxxxxxxx11000110111110b",   'c',
   "mull",       2,   "xxxxxxxxxx11000010111110b",   'c',
   "absf",       2,   "xxxxxxxxxx11010110111110b",   'c',
   "absl",       2,   "xxxxxxxxxx11010010111110b",   'c',

/* Format 4 instructions (12) */

   "add?",       2,   "xxxxxxxxxx0000iib",   'c',
   "cmp?",       2,   "xxxxxxxxxx0001iib",   'c',
   "bic?",       2,   "xxxxxxxxxx0010iib",   'c',
   "addc?",      2,   "xxxxxxxxxx0100iib",   'c',
   "mov?",       2,   "xxxxxxxxxx0101iib",   'c',
   "or?",        2,   "xxxxxxxxxx0110iib",   'c',
   "sub?",       2,   "xxxxxxxxxx1000iib",   'c',
   "addr",       2,   "xxxxxxxxxx100111b",   'c',
   "lxpd",       2,   "xxxxxxxxxx100111b",   'c',
   "and?",       2,   "xxxxxxxxxx1010iib",   'c',
   "subc?",      2,   "xxxxxxxxxx1100iib",   'c',
   "tbit?",      2,   "xxxxxxxxxx1101iib",   'c',
   "xor?",       2,   "xxxxxxxxxx1110iib",   'c',

/* Format 5 instructions (4) */

   "movst",      1,   "00000xxx100000ii00001110b",   'j',
   "movs?",      1,   "00000xxx000000ii00001110b",   'j',
   "cmpst",      1,   "00000xxx100001ii00001110b",   'j',
   "cmps?",      1,   "00000xxx000001ii00001110b",   'j',
   "skpst",      1,   "00000xxx100011ii00001110b",   'j',
   "skps?",      1,   "00000xxx000011ii00001110b",   'j',

   "setcfg",     1,   "00000xxxx000101100001110b",   'k',

/* Format 6 ops (14) */

   "rot?",       2,   "xxxxxxxxxx0000ii01001110b",   'c',
   "ash?",       2,   "xxxxxxxxxx0001ii01001110b",   'c',
   "cbit?",      2,   "xxxxxxxxxx0010ii01001110b",   'c',
   "cbiti?",     2,   "xxxxxxxxxx0011ii01001110b",   'c',
   "lsh?",       2,   "xxxxxxxxxx0101ii01001110b",   'c',
   "sbit?",      2,   "xxxxxxxxxx0110ii01001110b",   'c',
   "sbiti?",     2,   "xxxxxxxxxx0111ii01001110b",   'c',
   "neg?",       2,   "xxxxxxxxxx1000ii01001110b",   'c',
   "not?",       2,   "xxxxxxxxxx1001ii01001110b",   'c',
   "subp?",      2,   "xxxxxxxxxx1011ii01001110b",   'c',
   "abs?",       2,   "xxxxxxxxxx1100ii01001110b",   'c',
   "com?",       2,   "xxxxxxxxxx1101ii01001110b",   'c',
   "ibit?",      2,   "xxxxxxxxxx1110ii01001110b",   'c',
   "addp?",      2,   "xxxxxxxxxx1111ii01001110b",   'c',

/* Format 7 ops (15) */

   "movm?",      3,   "xxxxxxxxxx0000ii11001110b",   'o',
   "cmpm?",      2,   "xxxxxxxxxx0001ii11001110b",   'c',
   "inss?",      4,   "xxxxxxxxxx0010ii11001110b",   'i',
   "exts?",      4,   "xxxxxxxxxx0011ii11001110b",   'i',
   "movxbw?",    2,   "xxxxxxxxxx0100ii11001110b",   'c',
   "movzbw?",    2,   "xxxxxxxxxx0101ii11001110b",   'c',
   "movz?d",     2,   "xxxxxxxxxx0110ii11001110b",   'c',
   "movx?d",     2,   "xxxxxxxxxx0111ii11001110b",   'c',
   "mul?",       2,   "xxxxxxxxxx1000ii11001110b",   'c',
   "mei?",       2,   "xxxxxxxxxx1001ii11001110b",   'c',
   "dei?",       2,   "xxxxxxxxxx1011ii11001110b",   'c',
   "quo?",       2,   "xxxxxxxxxx1100ii11001110b",   'c',
   "rem?",       2,   "xxxxxxxxxx1101ii11001110b",   'c',
   "mod?",       2,   "xxxxxxxxxx1110ii11001110b",   'c',
   "div?",       2,   "xxxxxxxxxx1111ii11001110b",   'c',

/* Format 8 ops (8) */

   "ext?",       4,   "xxxxxxxxxxxxx0ii00101110b",   'm',
   "cvtp",       3,   "xxxxxxxxxxxxx01101101110b",   'm',
   "ins?",       4,   "xxxxxxxxxxxxx0ii10101110b",   'm',
   "check?",     3,   "xxxxxxxxxxxxx0ii11101110b",   'm',
   "index?",     3,   "xxxxxxxxxxxxx1ii00101110b",   'm',
   "ffs?",       2,   "xxxxxxxxxx0001ii01101110b",   'c',
   "movsu?",     2,   "xxxxxxxxxx0011ii10101110b",   'c',
   "movus?",     2,   "xxxxxxxxxx0111ii10101110b",   'c',

/* Format 9 ops (12) */

   "movlf",      2,   "xxxxxxxxxx0101ii00111110b",   'c',
   "movfl",      2,   "xxxxxxxxxx0111ii00111110b",   'c',
   "mov?f",      2,   "xxxxxxxxxx0001ii00111110b",   'c',
   "mov?l",      2,   "xxxxxxxxxx0000ii00111110b",   'c',
   "lfsr",       1,   "xxxxx000000011100111110b",   'a',
   "sfsr",       1,   "00000xxxxx11011100111110b",   'h',
   "roundf?",    2,   "xxxxxxxxxx1001ii00111110b",   'c',
   "roundl?",    2,   "xxxxxxxxxx1000ii00111110b",   'c',
   "truncf?",    2,   "xxxxxxxxxx1011ii00111110b",   'c',
   "truncl?",    2,   "xxxxxxxxxx1010ii00111110b",   'c',
   "floorf?",    2,   "xxxxxxxxxx1111ii00111110b",   'c',
   "floorl?",    2,   "xxxxxxxxxx1110ii00111110b",   'c',

/* Format 14 instructions (4) */

   "rdval",      1,   "xxxxxxxxx000001100011110b",   'a',
   "wrval",      1,   "xxxxxxxxx000011100011110b",   'a',
   "lmr",        2,   "xxxxxxxxx000101100011110b",   'e',
   "smr",        2,   "xxxxxxxxx000111100011110b",   'e'
};

   /* Address Mode Table */

#define MAXAM 42

struct {
   char *mstr;  /* mode match string */
   char *gstr;  /* output string to insert (gen) */
   int mcnt;    /* count of ambigs to be put into
               extension bytes */
   char mopt;   /* mode option */
} admode[ MAXAM ] = {

/* Scaled index modes */

   "*[r?:b]",   "11100",   1,   's',
   "*[r?:w]",   "11101",   1,   's',
   "*[r?:d]",   "11110",   1,   's',
   "*[r?:q]",   "11111",   1,   's',

/* Simple register modes */

   "r0",   "00000",   0,   ' ',   /* main registers */
   "r1",   "00001",   0,   ' ',
   "r2",   "00010",   0,   ' ',
   "r3",   "00011",   0,   ' ',
   "r4",   "00100",   0,   ' ',
   "r5",   "00101",   0,   ' ',
   "r6",   "00110",   0,   ' ',
   "r7",   "00111",   0,   ' ',

   "f0",   "00000",   0,   ' ',   /* floating point */
   "f1",   "00001",   0,   ' ',
   "f2",   "00010",   0,   ' ',
```

(continued on next page)

Listing One *(Listing continued)*

```
   "f3",    "00011",   0,    ' ',
   "f4",    "00100",   0,    ' ',
   "f5",    "00101",   0,    ' ',
   "f6",    "00110",   0,    ' ',
   "f7",    "00111",   0,    ' ',

/* Indexed addressing modes */

   "*(r0)",    "01000",   1,    ' ',   /* indexed */
   "*(r1)",    "01001",   1,    ' ',
   "*(r2)",    "01010",   1,    ' ',
   "*(r3)",    "01011",   1,    ' ',
   "*(r4)",    "01100",   1,    ' ',
   "*(r5)",    "01101",   1,    ' ',
   "*(r6)",    "01110",   1,    ' ',
   "*(r7)",    "01111",   1,    ' ',

   "*(*(fp))",  "10000",   2,    'r',   /* frame ptr */
   "*(*(sp))",  "10001",   2,    'r',   /* stack mem */
   "*(*(sb))",  "10010",   2,    'r',   /* static mem */

   "#*",        "10100",   1,    ' ',   /* immediate */
   "@*",        "10101",   1,    ' ',   /* absolute */
   "ext(*)+*",  "10110",   2,    ' ',   /* external */
   "tos",       "10111",   0,    ' ',   /* top of stack */

   "*(fp)",     "11000",   1,    ' ',   /* frame mem */
   "*(sp)",     "11001",   1,    ' ',   /* stack mem */
   "*(sb)",     "11010",   1,    ' ',   /* static mem */

   ".+*",       "11011",   1,    ' ',   /* program mem */

   "[*]",       "",   0,   'l',   /* register list */

/* catch-all */

   "*",   "",   1,   'w' /* fits no pattern */
};

/*---MAIN PROGRAM---*/

main( argc, argv )
int argc;
char *argv[];
{
   int i;

   puts( "\nA32000 v0.10" );

   if( argc < 2 ) {
      puts( "\n?No file name specified" );
      exit( 1 );
   }

   symcnt = 0;

   for( pass = 1; pass <= 3; ++pass ) {

      makename( argv[ 1 ], ".s" );
      fasm = fopen( filename, "r" );

      if( fasm == 0 ) {
         puts( "\n?Unable to open source file" );
         exit( 1 );
      }

      if( pass == 3 ) {
         makename( argv[ 1 ], ".hex" );
         fobj = fopen( filename, "w" );
         if( ! fobj ) {
            puts( "\n?No directory space" );
            exit( 1 );
         }
      }

      puts( "\nPass " );
      putchar( pass + '0' );

      asmadr = 0;
      codadr = 0;
      if( pass == 3 ) {
```

```
        objflush();
        listnl();
     }
     inpload();

     while( gword() ) {

        if( match( word, "end" )) break;

/* Each word is processed by the following nested if
   statement, which attempts to identify what it is.
   Note that any successful identification stops the
   process of the statement. */

        if( ! islabel( word ))
           if( ! ispseudo( word ))
              if( ! isopcode( word ))
                 if( ! isequate( word ))
                    error( '?', word );
     }

     fclose( fasm );

/* Sort symbols after pass 1. */

     if( pass == 1 ) sortsyms();

     if( pass == 3 )   {
        objflush();

        putc( ':', fobj );     /* write eof record */
        for( i = 0; i < 10; ++i ) putc( '0', fobj );
        putc( '\n', fobj );

        fclose( fobj );
     }
  }

  listpr();
  puts( "\n\n" );
  dumpsyms();

  if( errors )
     puts( "\n---Fix errors and reassemble---" );
}

/* Construct a filename from two strings. */

makename( p, q )
char *p, *q;
{
   char *r;
   r = &filename[ 0 ];
   while( *p ) *r++ = *p++;
   while( *q ) *r++ = *q++;
   *r = '\0';
}

/* Check to see if the word is a label, and if it is, add
   its value to the symbol table */

int islabel( w )
char *w;
{
   while( *w ) ++w;
   if( *--w != ':' ) return 0;
   *w = '\0';    /* take off the colon */

   addsymbol( word, codadr );
   return 1;
}

/* Check the word to see if it is a pseudo-op. */

int ispseudo( w )
char *w;
{
   long int getarg(), temp;

   if( match( w, "org" )) {
   asmadr = getarg();
   codadr = asmadr;
   if( pass == 3 ) objflush();
```

(continued on next page)

Listing One *(Listing continued)*

```
      return 1;
   }

   if( match( w, "db" )) {       /* Note: Allow msgs? */
      temp = getarg();           /* get argument */
      objout( temp & 0xFF );     /* output byte */
      return 1;
   }

   if( match( w, "dw" )) {
      temp = getarg();           /* get argument */
      objout( temp & 0xFF );     /* output lsb */
      objout(( temp >> 8 ) & 0xFF );     /* output msb */
      return 1;
   }

   if( match( w, "dd" )) {
      temp = getarg();           /* get argument */
      objout( temp & 0xFF );     /* output lsb */
      objout(( temp >> 8 ) & 0xFF );
      objout(( temp >> 16 ) & 0xFF );
      objout(( temp >> 24 ) & 0xFF );    /* output msb */
      return 1;
   }

   if( match( w, "even" ) && ( codadr & 1 )) {
      objout( 0 ); /* send 1 byte to go to word bndry */
      return 1;
   }

   return 0;
}

/* Check to see if the word is an opcode, and if it is,
   get any operands required and generate code. */

int isopcode( w )
char *w;
{
   long int value(), bitbin(), decbin(), o, ocodadr;
   char opbuf[ 33 ], bytbuf[ 33 ], extbuf[ 128 ];
   char opopt, modopt, opsiz, opcnt;
   int i, j, k, l;
   char *p, *q, *cpystr(), *regbits();

   /* postbytes & scaled indexes */
   int opexbt[ 4 ], opexct;

   int adexct, adexln[ 8 ];
   char *adexpt[ 8 ], *eoadex; /* addressing extensions */

   ocodadr = codadr;    /* save addr of begin of instr */

   opexct = 0;          /* no postbytes as yet */
   adexct = 0;          /* no extensions as yet */
   eoadex = &extbuf[ 0 ]; /* point to begin of extbuf */

   for( i = 0; i < MAXOP; ++i )
      if( match( w, opcode[ i ].onam )) {

      opopt = opcode[ i ].oopt;

      if( opopt == 'x' ) {
         error( 'x', w ); /* unimplemented instruction */
         return 1;
      }

      p = cpystr( opcode[ i ].obin, &opbuf[ 0 ] );

/* see if length modifier */

      if( ambcnt > 0 ) {
         p = &opbuf[ 0 ];
         opsiz = *ambig[ 0 ];

         while( *p && *p != 'i' ) ++p;

         if( ! *p ) error( 'l', w );
         else {
         switch( opsiz ) {

            case 'b' : *p++ = '0';
               *p++ = '0';
               break;

            case 'w' : *p++ = '0';
               *p++ = '1';
               break;

            case 'd' : *p++ = '1';
               *p++ = '1';
            }
         }
      }

/* now parse operands */

/* get count of operands.
   Take abs value (neg = PC-relative) */

      opcnt = opcode[ i ].ocnt;
      if( opcnt < 0 ) opcnt = 0 - opcnt;

      p = &opbuf[ 0 ];  /* modified parts start at beg. */

      for( j = 0; j < opcnt; ++j ) {
         gword();       /* get operand */

/* find addr mode */

         k = 0;
         while(( k < MAXAM )
            && ! match( word, admode[ k ].mstr )) ++k;

         modopt = admode[ k ].mopt;

/* move bit string into place */

         q = admode[ k ].gstr;

/* if opopt h, sfsr, skip 5 bits */

         if( opopt == 'h' ) p += 5;

/* for most opopts, move the bits in */

         if(( opopt == ' ' )
            || ( opopt == 'a' )
            || ( opopt == 'c' )
            || ( opopt == 'e' && j == 1 )
            || ( opopt == 'h' )
            || ( opopt == 'i' && j < 2 )
            || ( opopt == 'l' && j == 1 )
            || ( opopt == 'm' && j > 0 && j < 3 )
            || ( opopt == 'o' && j < 2 ))
            while( *q ) *p++ = *q++;

/* Double the effort for scaled index mode.  create an
   extension postbyte opexbt[] with basemode as upper
   5 bits, reg as lower 3 bits. */

         if( admode[ k ].mopt == 's' ) {
            l = ( *ambig[ 1 ] ) & 7;
            q = cpystr( ambig[ 0 ], &bytbuf[ 0 ] );

/* find basemode */

            k = 0;
            while(( k < MAXAM )
               && ! match( &bytbuf[ 0 ],
               admode[ k ].mstr )) ++k;

            modopt = admode[ k ].mopt;

/* move bit string into postbyte.  use bitbin, because
   value() destroys ambig[] array which we still need. */

            q = cpystr( admode[ k ].gstr,
               &bytbuf[ 0 ] );
            --q;                 /* back up to null */
            *q++ = '0' + (( l >> 2 ) & 1 );
            *q++ = '0' + (( l >> 1 ) & 1 );
```

```
            *q++ = '0' + ( l & 1 );
            *q = '\0';

            opexbt[ opexct++ ] = bitbin( &bytbuf[ 0 ] );
            q = admode[ k ].gstr;
         }

/* funny handling of reg: index operation (opopt 'm') */

         if( opopt == 'm' && j == 0 ) {
            p = &opbuf[ 10 ];    /* off to reg */
            q += 2;              /* skip 0 bits */
            while( *q ) *p++ = *q++;
            p = &opbuf[ 0 ];     /* reset */
         }

/* move ambigs into extension bytes. set length to
   variable (0). For some addressing modes, the
   extensions go in in reverse order */

         if( modopt == 'r' )
            for( l = admode[ k ].mcnt - 1; l >= 0;
               --l ) {
               adexpt[ adexct ] = eoadex;
               adexln[ adexct ] = 0;
               ++adexct;
               eoadex = cpystr( ambig[ l ], \
                  eoadex );
         } else for( l = 0; l < admode[ k ].mcnt; ++l ) {
               adexpt[ adexct ] = eoadex;
               adexln[ adexct ] = 0;
               ++adexct;
               eoadex = cpystr( ambig[ l ], \
                  eoadex );
         }

/* special logic for register list for "enter", "save",
   "restore", "exit" */

         if(( j == 0 && opopt == 'f' ) || opopt == 'g' ) {
            adexpt[ adexct ] = eoadex;
            adexln[ adexct ] = 1;
            ++adexct;
            eoadex = regbits( word, eoadex, opopt );
         }

/* shorten extension to 1 byte for enter, return */

         if(( j == 1 && opopt == 'f' ) || opopt == 'n' ) {
            if( adexct > 0 ) adexln[ adexct - 1 ] = 1;
               else error( 'e', w );
         }

/* opopts 'e' or 'd': immed data becomes 4 bit value */

         if(( j == 0 && opopt == 'e' ) || opopt == 'd' ) {
            if( ! adexct ) error( 'e', w );
               else {
               l = value( adexpt[ --adexct ] );
               p = &opbuf[ 5 ];
               *p++ = '0' + (( l >> 3 ) & 1 );
               *p++ = '0' + (( l >> 2 ) & 1 );
               *p++ = '0' + (( l >> 1 ) & 1 );
               *p = '0' + ( l & 1 );
               p = &opbuf[ 0 ];
            }
         }

/* opopt 'i': combine last two ambigs into one postbyte.
   Put it in bytbuf and set length to 1 */

         if( opopt == 'i' && j == 3 ) {
            if( adexct < 2 ) error( 'e', w );
            else {
               l = ( value( adexpt[ --adexct ] ) - 1 )
                  & 31;
               l += ( value( adexpt[ --adexct ] )
                  & 7 ) << 5;
               bytbuf[ 0 ] = '0' +
                  (( l / 100 ) % 10 );
               bytbuf[ 1 ] = '0' +
                  (( l / 10 ) % 10 );
```

(continued on next page)

Listing One *(Listing continued)*

```
            bytbuf[ 2 ] = '0' + ( l % 10 );
            bytbuf[ 3 ] = '\0';
            adexpt[ adexct ] = &bytbuf[ 0 ];
            adexln[ adexct++ ] = 1;
         }
      }

/* opopt 'j': uwb bits for movs, cmps, skps */

      if( opopt == 'j' ) {
         p += 5;
         uwbbits( word, p );
         adexct = 0;          /* in case no paren */
      }

/* opopt 'k': config bits for setcfg */

      if( opopt == 'k' ) {
         p += 5;
         cfgbits( word, p );
      }

/* opopt 'l': operand 1 becomes 4 bit value */

      if( opopt == 'l' && j == 0 ) {
         adexct = 0;          /* unjunk extensions */
         l = -1;
         if( strcmp( word, "upsr" ) == 0 ) l = 0;
         if( strcmp( word, "fp" ) == 0 ) l = 8;
         if( strcmp( word, "sp" ) == 0 ) l = 9;
         if( strcmp( word, "sb" ) == 0 ) l = 10;
         if( strcmp( word, "psr" ) == 0 ) l = 13;
         if( strcmp( word, "intbase" ) == 0 ) l = 14;
         if( strcmp( word, "mod" ) == 0 ) l = 15;
         if( l == -1 ) error( 'p', word );
            else {
            p = &opbuf[ 5 ];
            *p++ = '0' + (( l >> 3 ) & 1 );
            *p++ = '0' + (( l >> 2 ) & 1 );
            *p++ = '0' + (( l >> 1 ) & 1 );
            *p = '0' + ( l & 1 );
            p = &opbuf[ 0 ];
            }
      }

/* odd length extension for movm, opopt 'o' */

      if( j == 2 && opopt == 'o' ) {
         if( adexct > 0 ) adexln[ --adexct ] = 1;
            else error( 'e', w );
         l = value( adexpt[ adexct ] ) - 1;
         switch( opsiz ) {
         case 'd' : l *= 4; break;
         case 'w' : l *= 2; break;
         }
         bytbuf[ 0 ] = '0' + (( l / 100 ) % 10 );
         bytbuf[ 1 ] = '0' + (( l / 10 ) % 10 );
         bytbuf[ 2 ] = '0' + ( l % 10 );
         bytbuf[ 3 ] = '\0';
         adexpt[ adexct++ ] = &bytbuf[ 0 ];
      }

   }        /* done operands */

   o = value( &opbuf[ 0 ] );

   l = strlen( opbuf );

/* Send as many opcode bytes as necessary */

   objout( o % 256 );
   if( l > 9 ) objout(( o / 256 ) % 256 );
   if( l > 17 ) objout( o / 65536 );

/* Send postbytes for scaled index mode */

   for( l = 0; l < opexct; ++l )
      objout( opexbt[ l ] );

/* Send addressing extensions.
```

```
   adexln = length of extension word in bytes if +.
   If 0, it is a variable-length signed displacement.
   If -1, indicates code-relative */

/* If opcode ocnt was negative, last address extension is
   code relative. */

      if( opcode[ i ].ocnt < 0 )
         adexln[ adexct - 1 ] = -1;

/* send extension words */

      for( j = 0; j < adexct; ++j ) {

         o = value( adexpt[ j ] );

/* if adexln[] negative, operand(s) code-relative.
   Note: on the 32000 you don't correct by adding 2 to
   codadr first */

         if( adexln[ j ] < 0 ) o -= ocodadr;

/* Compute variable-length signed displacement */

         if( adexln[ j ] <= 0 ) {
            if( o < 63 && o > -64 ) {
               o = ( o & 0x7F );
               l = 1;           /* one-byte */
            } else if( o < 8191 && o > -8192 ) {
               o = ( o & 0x3FFF ) + 0x8000;
               l = 2;
            } else {
               o = ( o & 0x3FFFFFFF )
                  + 0xC0000000;
               l = 4;
            }
         } else l = adexln[ j ];

/* address extensions are sent in lohi order */

         if( l > 3 ) objout(( o >> 24 ) & 0xFF );
         if( l > 2 ) objout(( o >> 16 ) & 0xFF );
         if( l > 1 ) objout(( o >> 8 ) & 0xFF );
         objout( o % 256 );
      }
      return 1;
   }
   return 0;
}

/* Special to create extension word for register list */

/* Regbits may look like "r0" or may look like "[r0,r2]"
   or like "[r0-r7]". */

char *regbits( src, dst, flg )
char *src, *dst, flg;
{
   int bits, reg, loreg, hireg;

   bits = 0;

   if( *src == '[' ) ++src;      /* strip parens */
      else error( '[', src );

   while( *src ) {
      if( *src++ != 'r' ) error( 'r', src );
      reg = ( *src++ ) - '0';
      bits = bits | ( 1 << reg );
      if( *src++ == '-' ) {
         loreg = reg;
         if( *src++ != 'r' ) error( 'r', src );
         hireg = ( *src++ ) - '0';
         if( hireg < loreg ) {
            reg = hireg;
            hireg = loreg;
            loreg = reg;
         }              /* swap if out of order */
         for( reg = loreg; reg <= hireg; ++reg )
            bits = bits | ( 1 << reg );
         ++src;         /* skip over the comma */
      }
      if( *src == ']' ) break;
```

(continued on next page)

Listing One *(Listing continued)*

```
   }

/* if flg = 'f', save/enter, need to swap bit
   significance. The routine above constructed it in
   reversed order in the first place, because of the
   routine below */

   if( flg == 'f' ) {
      hireg = 0;
      for( reg = 0; reg < 8; ++reg ) {
         hireg = ( hireg << 1 ) + ( bits & 1 );
         bits = ( bits >> 1 );
      }
      bits = hireg;
   }

/* now create a binary string for the extension.
   Note that bit significance becomes reversed again */

   for( reg = 0; reg < 8; ++reg ) {
      *dst++ = '0' + ( bits & 1 );
      bits = ( bits >> 1 );
   }

   *dst++ = 'b';         /* add b for binary */
   *dst++ = '\0';        /* terminate */

   return dst;
}

/* put config bits into instruction */

cfgbits( src, dst )
char *src, *dst;
{
   char b[ 4 ];

   b[ 0 ] = '0';
   b[ 1 ] = '0';
   b[ 2 ] = '0';
   b[ 3 ] = '0';

   if( *src == '[' ) ++src;      /* strip parens */
      else error( '[', src );

   while( *src ) {
      switch( *src++ ) {

      case 'c' : b[ 0 ] = '1';
            break;

      case 'm' : b[ 1 ] = '1';
            break;

      case 'f' : b[ 2 ] = '1';
            break;

      case 'i' : b[ 3 ] = '1';
      }

      if( *src++ == ']' ) break;
   }
   *dst++ = b[ 0 ];
   *dst++ = b[ 1 ];
   *dst++ = b[ 2 ];
   *dst = b[ 3 ];
}

/* put uwb bits into instruction */

uwbbits( src, dst )
char *src, *dst;
{
   char b[ 3 ];

   b[ 0 ] = '0';         /* default = forward */
   b[ 1 ] = '0';         /* default = neither */
   b[ 2 ] = '0';

   while( *src ) {
      switch( *src++ ) {

      case 'b' : b[ 2 ] = '1';  /* backward */
            break;

      case 'u' : b[ 0 ] = '1';  /* until match */
            b[ 1 ] = '1';
            break;

      case 'w' : b[ 0 ] = '0';  /* while match */
            b[ 1 ] = '1';
      }
   }
   *dst++ = b[ 0 ];
   *dst++ = b[ 1 ];
   *dst = b[ 2 ];
}

/* Check to see if the word begins an equate, and if it
   does, add the symbol to the symbol table. */

int isequate( w )
char *w;
{
   char tempword[ 128 ];
   char *q, *cpystr();
   long int l, getarg();

   q = cpystr( w, &tempword[ 0 ] );

   gword();               /* get next word */

   if( strcmp( word, "equ" ) == 0 ||
      strcmp( word, "=" ) == 0 ) {
      l = getarg();       /* get argument */

      addsymbol( &tempword[ 0 ], l );

      return 1;           /* it was an equate */
   }

   return 0;              /* we lost a word */
}

/* Get an argument value (for use above). */

long int getarg()
{
   long int value();

   gword();               /* get next word */
   return value( word );
}

/* copy string and return new ending address */

char *cpystr( src, dst )
char *src, *dst;
{
   while( *src ) *dst++ = *src++;
   *dst++ = '\0';         /* terminate copied string */
   return dst;            /* return next address */
}

/* Calculate the value of a word.  It may be a symbol, a
   constant, or a computed value (must be enclosed in
   parentheses.) */

long int value( w )
char *w;
{
   long int hexbin(), octbin(), bitbin(), decbin(), v;
   int lookup(), i, negate;

   char *q;

   char *wp[ 16 ];

   int wpcnt;

   negate = 0;

   if( *w == '-' ) {     /* Unary negation */
```

```
        negate = 1;
        ++w;
    }

    if( strcmp( w, "." ) == 0 )
        return codadr; /* . = code address */
    if( strcmp( w, ".." ) == 0 )
        return asmadr; /* .. = assembly address */

    if( isdigit( *w )) {
        if( match( w, "*h" )) v = hexbin( w );
            else if( match( w, "*q" )) v = octbin( w );
                else if( match( w, "*b" ))
                    v = bitbin( w );
                    else v = decbin( w );
    } else {

        if( *w == '(' ) {           /* --- FORMULA --- */
            ++w;                    /* skip ( */
            q = w;
            while( *q ) ++q;        /* find end of string */
            --q;
            if( *q != ')' ) error( ')', q );
                else *q = '\0';     /* zap ) */

            iwparen = 0;            /* no parens now */

            wpcnt = 0;

            while( 1 ) {            /* find beg of word */
                while( inword( *w )) ++w;

                if( ! *w ) break;

                wp[ wpcnt++ ] = w;  /* ptr to value */

                iwparen = 0;        /* find end of word */
                while( *w && ! inword( *w )) ++w;
                if( ! *w ) break;
                *w++ = '\0';        /* terminate it */

                if( wpcnt == 16 ) {
                    error( 'l', w ); /* too long */
                    break;
                }
            }

            if(( wpcnt % 2 ) == 0 ) {
                error( 'v', w );    /* must be odd */
                --wpcnt;
            }

            v = value( wp[ 0 ] );

            for( i = 1; i < wpcnt; i += 2 ) {
                if( strcmp( wp[ i ], "+" ) == 0 ) {
                    v += value( wp[ i + 1 ] );
                    goto opdone;
                }
                if( strcmp( wp[ i ], "-" ) == 0 ) {
                    v -= value( wp[ i + 1 ] );
                    goto opdone;
                }
                if( strcmp( wp[ i ], "*" ) == 0 ) {
                    v *= value( wp[ i + 1 ] );
                    goto opdone;
                }
                if( strcmp( wp[ i ], "/" ) == 0 ) {
                    v /= value( wp[ i + 1 ] );
                    goto opdone;
                }
                error( 'o', wp[ i ] );      /* unknown op */
opdone:
                i = i;              /* get around c/80 bug */
            }
        } else {                    /* --- PLAIN VALUE --- */

            i = lookup( w );        /* look up symbol */
            if( i < 0 ) return 0;   /* unknown symbol */
            v = symbol[ i ].sval;   /* return sym value */
        }
    }
```

(continued on next page)

Listing One *(Listing continued)*

```
   if( negate ) v = 0 - v;

   return v;
}

/* function for value() */

int inword( c )
char c;
{
   if( c == '(' ) ++iwparen;    /* special var for this */
   if( c == ')' ) --iwparen;    /* function */

   if( iwparen ) return 0;

   if( c == ' ' ) return 1;     /* is space */

   return 0;
}

/* --- SYMBOL TABLE LOGIC --- */

/* add new symbol to symbol table */

addsymbol( p, v )
char *p;
long int v;
{
   char *w, *cpystr(), *alloc();
   int i, lookup();

   i = lookup( p );      /* see if already known */

   if( i < 0 ) {         /* new symbol */
      i = symcnt;
      ++symcnt;          /* count a new symbol */

      symbol[ i ].snam = alloc( strlen( p ) + 1 );
      w = cpystr( p, symbol[ i ].snam );
   }

   symbol[ i ].sval = v; /* update value in table */
}

/* lookup - returns symbol number or -1 if not found */

int lookup( p )
char *p;
{
   char *w;
   int i, j, k, found;

   found = 0;            /* not found yet */

/* pass 1 - use linear search */

   if( pass == 1 ) {
      for( i = 0; i < symcnt && ! found; ++i ) {
         w = symbol[ i ].snam;
         found = ( strcmp( p, w ) == 0 );
      }
   } else {

/* passes 2 and 3 - use binary search */

      j = ( symcnt + 1 ) / 4;    /* step to use */
      i = symcnt / 2;            /* starting point */
      k = j + 1;                 /* one-step count */

      while( 1 ) {
         w = symbol[ i ].snam;
         found = strcmp( p, w );
         if( found == 0 ) {
            found = 1;
            break;
         } else if( found < 0 ) i -= j;
           else i += j;

         if( i < 0 ) i = 0;
         if( i >= symcnt ) i = symcnt - 1;

         j /= 2;                 /* halve step */

         if( j == 0 ) {
            if( k-- == 0 ) {
               found = 0;        /* not found */
               break;
            }
            j = 1;
         }
      }
   }

   if( ! found ) {
      if( pass != 1 ) error( 'u', w );
      return -1;
   }

   return i;
}

/* display error code */

error( c, p )
char c;
char *p;
{
   puts( "\n>>---> Error " );
   putchar( c );
   puts( " at " );
   puts( p );

   ++errors;
}

/* sort symbols by shell sort */

sortsyms()
{
   int jump, done, k, l;
   char *n;
   long int v;

   jump = symcnt;        /* set jmp to cnt of elements */

   while( jump > 0 ) {
      jump = jump / 2;

      while( 1 ) {
         done = 1;
         for( k = 0; k < ( symcnt - jump ); ++k ) {
            l = k + jump;
            if( strcmp( symbol[ k ].snam,
               symbol[ l ].snam ) > 0 ) {
               n = symbol[ k ].snam;
               v = symbol[ k ].sval;
               symbol[ k ].snam = symbol[ l ].snam;
               symbol[ k ].sval = symbol[ l ].sval;
               symbol[ l ].snam = n;
               symbol[ l ].sval = v;
               done = 0;
            }
         }
         if( done ) break;
      }
   }
}

/* dump symbol table */

dumpsyms()
{
   char *w;
   int i;
   long int v;

   puts( "\nSymbol and Value\n" );

   for( i = 0; i < symcnt; ++i ) {
      puts( symbol[ i ].snam );
      puts( " = " );
      v = symbol[ i ].sval;
      puthex( v >> 24, 0 );
      puthex(( v >> 16 ) & 0xFF, 0 );
```

(continued on page 97)

Listing One *(Listing continued)*

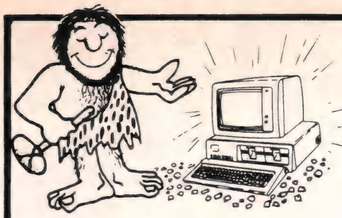
```
      puthex(( v >> 8 ) & 0xFF, 0 );
      puthex( v & 0xFF, 0 );    /* print value */
      putchar( '\n' );
   }
}

/* Match string.  If match, returns 1; else returns 0.
   Ambiguous values from the matches are saved and
   pointed to by the array of char pointers ambig[], so
   they can be checked later. */

int match( w1, w2 )
char *w1, *w2;
{
   char c;
   char *next_ambig;

   next_ambig = &ambig_buffer[ 0 ]; /* init ambig buff */
   ambcnt = 0;                      /* ambigs so far */

   while( *w1 ) {
      c = *w2++;
      if( c == '*' ) {
         ambig[ ambcnt++ ] = next_ambig;
         while( *w1 && *w1 != *w2 )
            *next_ambig++ = *w1++;
         if( ! *w1 && *w2 ) return 0;
         *next_ambig++ = '\0';  /* terminate this ambig */
      } else if( c == '?' ) {
           ambig[ ambcnt++ ] = next_ambig;
           *next_ambig++ = *w1++;      /* 1-char ambig */
           *next_ambig++ = '\0';       /* terminate it */
      } else if( c != *w1++ ) return 0;
   }
   return 1;
}

int gword()
{
   char *p, *q;
   char c, gchar();

   p = &word_buffer[ 0 ];

   c = ' ';

   while( isdelim( c )) c = gchar();

   while( ! isdelim( c )) {
      *p++ = tolower( c );
      c = gchar();
   }

   *p = '\0';   /* terminate word */

   word = &word_buffer[ 0 ];

   return 1;
}

/* is the character a delimeter? */

isdelim( c )
char c;
{
   if( paren || quote || brack )
      return 0; /* not a delim */

   if( c == ' ' || c == ',' || c == ';' || c == '\n' \
      || c == '\r' || c == '\t' )
      return 1;
   return 0;
}

/* get next char from source file */

char gchar()
{
   char c, getch();

   c = getch();          /* get char from file */
   if( c == '\'' ) quote = ! quote;
   if( c == '"' ) quote = ! quote;
   if( c == '(' ) ++paren;
   if( c == ')' ) --paren;
   if( c == '[' ) ++brack;
   if( c == ']' ) --brack;

   if( ! quote && ! paren && ! brack ) {

      while( c == ';' ) {       /* ;comment\n */
         while( getch() != '\n' ) ;
         c = getch();
      }
   }

   return c;
}

puts( p )
char *p;
{
   while( *p ) putchar( *p++ );
}

/* --- source file routines --- */

char getch()
{
   while( inpcnt == 0 ) {       /* if input buf empty, */
      listpr();          /* print listing line */
      inpload();         /* reload input buffer */
   }

   --inpcnt;
   return( inpbuf[ inpptr++ ] );
}

inpload()
{
   char c, getc();

   inpcnt = 0;
   inpptr = 0;

   while((( c = getc( fasm )) != '\n' ) && ( c != EOF )) {
      inpbuf[ inpcnt++ ] = c;
      if( listcp < 81 ) listline[ listcp++ ] = c;
   }

   inpbuf[ inpcnt++ ] = '\n';
}

/* --- listing file routines --- */

listnl()        /* list new line */
{
   int i;

   if( pass != 3 ) return;

   for( i = 0; i < 26; ++i ) listline[ i ] = ' ';
   for( i = 26; i < 81; ++i ) listline[ i ] = '\0';

   listop = 0;  /* flag to cause addr output */
   listcp = 26;
}

lbyt( b )       /* put object byte in list file */
unsigned int b;
{
   char c;

   if( pass != 3 ) return;

   c = (( b / 16 ) % 16 ) + '0';
   if( c > '9' ) c += ( 'a' - ':' );
   listline[ listop++ ] = c;

   c = ( b % 16 ) + '0';
   if( c > '9' ) c += ( 'a' - ':' );
   listline[ listop++ ] = c;
```

(continued on next page)

Listing One *(Listing continued)*

```
   if( listop > 24 ) listpr();  /* print list line */
}

listpr()         /* print list line */
{
   if( pass != 3 ) return;

   putchar( '\n' );
   puts( listline );
   listnl();
}

/* --- object file routines --- */

objout( c )
char c;
{
   asmadr++;    /* incr asmadr, codadr. DON'T incr*/
   codadr++;    /* objadr, it is addr of 1st byte */
   if( pass != 3 ) return;   /* skip if not last pass */
   objbuf[ objcnt++ ] = c;   /* put new byte in buffer */
   if( objcnt == 32 ) objflush();

   if( listop == 0 ) {           /* print address? */
      lbyt( asmadr / 16777216 );
      lbyt(( asmadr / 65536 ) % 256 );
      lbyt(( asmadr / 256 ) % 256 );
      lbyt( asmadr % 256 );
      listop = 9;
   }
   lbyt( c );            /* send byte to listing too */
}

objflush()
{
   int i, cksum;

   if( pass != 3 ) return; /* just in case we get here */

   cksum = 0;

   if( objcnt > 0 ) {
      putc( ':', fobj );
      puthex( objcnt, fobj );
      puthex( objadr / 256, fobj );
      puthex( objadr % 256, fobj );
      puthex( 0, fobj );
      cksum =
         objcnt + ( objadr / 256 ) + ( objadr % 256 );
      for( i = 0; i < objcnt; ++i ) {
         puthex( objbuf[ i ], fobj );
         cksum += objbuf[ i ];
      }
      puthex( 0 - cksum, fobj );
      putc( '\n', fobj );
   }

   objadr = asmadr;
   objcnt = 0;}

puthex( b, c )
int b, c;
{
   int v;

   v = ( b & 0x00F0 ) >> 4;
   if( v > 9 ) v += 'A' - 10; else v += '0';
   putc( v, c );
   v = ( b & 0x000F );
   if( v > 9 ) v += 'A' - 10; else v += '0';
   putc( v, c );
}

long int hexbin( p )
char *p;
{
   long int v;

   v = 0;

   while( *p ) {
      if( isdigit( *p )) v = ( 16 * v ) + *p++ - '0';
         else if( *p >= 'a' && *p <= 'f' )
            v = ( 16 * v ) + *p++ - 'a' + 10;
            else ++p;
   }

   return v;
}

long int octbin( p )
char *p;
{
   long int v;

   v = 0;

   while( *p ) {
      if( *p >= '0' && *p <= '7' )
         v = ( 8 * v ) + *p++ - '0';
         else ++p;
   }
   return v;
}

long int bitbin( p )
char *p;
{
   long int v;

   v = 0;

   while( *p ) {
      if( *p == '0' || *p == '1' )
         v = ( 2 * v ) + *p++ - '0';
         else ++p;
   }
   return v;
}

long int decbin( p )
char *p;
{
   long int v;

   v = 0;

   while( *p ) {
      if( isdigit( *p )) v = ( 10 * v ) + *p++ - '0';
         else ++p;
   }
   return v;
}

#include "stdlib.c"
```

End Listing



From B C Associates — SimpleNET™

SIMPLY THE LOWEST COST LAN

Yes, it's true, now you can take control of your data handling problems and implement your own PC local area network for only **$99.00** per station (plus software, power supply and cables).

Are you tired of carrying around a box of diskettes just to transfer information among your many PC systems in your office? You've probably looked into networks, but the high cost of such systems kept you away.

Now there's SimpleNET™. A truly low cost/medium performance alternate to the high priced systems. SimpleNET uses a small interface module which attaches to your PC systems and a PC Network (DOS 3.x) compatible network BIOS program. The interface allows up to 32 users to be connected via a single interface cable with a maximum cable length of 1.2 kilometers (how about 4000 feet?). The software interface is compatible with DOS 3.x and the new PC Local Area Network Program available from your IBM dealer.

SimpleNET Basic System* — For up to 4 users

- Interface/Power supply module. (One power supply module is capable of driving 8 stations.)
- User Interface modules
- Cable Package
- Network BIOS Software
- Installation/Operations Manual

Only $695.00 Complete

Additional user can be added for $99.00 each.

*Requires IBM PC/XT/AT or compatible with one available asynchronous communications port.

PROGRAMMERS AND SOFTWARE DEVELOPERS - LOOK AT THESE PRODUCTS! NO ROYALTIES REQUIRED

ASMLIB
The Programmer's Library

- A Multipurpose set of over 200 Assembly Language sub routines supplied in the form of a linkable library.
- Virtual disk file handling.
- Int. driven asynch. support.
- Graphics on EGA, herc. and CGA.
- Floating point math and trig routines with 8087 support.
- Installable keyboard activated programs are easily written with ASMLIB's special functions.
- Plus much, much more.
- Supplied with complete source code.

Now with MSC v4.00 support !

Only $149.00 Complete

asmTREE
The Programmer's B+Tree Data File Management System

- A complete single/ multiuser database management system written entirely in Assembly Language gives the Lattice "C" or Assembly Language programmer these capabilities.
- Up to 256 users.
- Up to 256 index and data files.
- Multiple key types.
- Multiple indices per index file.
- Duplicate and variable length keys.
- Virtual file handling
- Plus much, much more
- Supplied with complete source code.

Now with MSC v4.00 support !

Only $395.00 Complete

GenericGL - Generic general ledger package can be used by any program...UDS...$295.00

FSEdit - Full sreen edit package by UDS...$49.95

REALIA COBOL USERS!

FPLIB - Floating point library package with 8087 support and trig functions...$149.00

Full Money Back Guarantee

B C ASSOCIATES
3261 No. Harbor Blvd., Suite B
Fullerton, CA 92635
1-800-262-8010
in Calif. Call
(714) 526-5151

Enclosed please find my ☐Check ☐Money order for $ ____________

Please send the following:

QTY	Item	Price	
____	SimpleNET Basic 4 user...by UTE	$695.00 each =	____
____	asmTREE database development system	$395.00 each =	____
____	ASMLIB Assembly Language library	$149.00 each =	____
____	GenericGL general ledger package	$295.00 each =	____
____	FSEdit full screen editor	$ 49.95 each =	____
____	FPLIB Realia COBOL Floating point pkg	$149.00 each =	____

All prices include UPS shipping within continental United States. Outside U.S. please add $10 per package. Calif. residents please add 6.5% sales tax. = ____

VISA MasterCard

Total = ____

Listing One *(Text begins on page 104.)*

```
#include <stdio.h>
#include <fcntl.h>
#include <getargs.h>

/*      EXEPRINT.C      Either print or modify the exe file header:
 *
 *      exe file        Print the contents of a file's EXE header
 *      exe -mN file    Modify the exe header so that N bytes of memory
 *                      are allocated for the combined bss/stack/heap
 *                      area. If N is smaller that the required minimum
 *                      (bss + stack size) then it's rounded up. The
 *                      largest permitted value of N is 65,535. Use
 *                      -m1 for the minimum possible heap.
 *      exe -sN file    Modify the exe header so that N bytes of stack
 *                      are used. If necessary, increase the bss/stack/heap
 *                      size to accommodate the new stack. (The
 *                      bss/stack/heap won't be made smaller, however).
 */

typedef unsigned short  word;   /* 2-byte unsigned number               */

typedef struct
{
        word    signature;
        word    image_len;      /* Length of load module image % 512      */
        word    file_size;      /* File size in 512-byte units            */
        word    num_reloc;      /* Number of relocation table items       */
        word    header_size;    /* Size of the neader in paragraphs       */
        word    bss_min;        /* min size of data area above program    */
        word    bss_max;        /* max size of data area above program    */
        word    stack_disp;     /* displacement in para. of stack seg.    */
        word    init_sp;        /* Initial SP register contents           */
        word    checksum;       /* Checksum for file                      */
        word    init_ip;        /* Initial IP register contents (PC)      */
        word    code_disp;      /* displacement in para. to code seg.     */
        word    first_reloc;    /* displacement (bytes) to 1st reloc item */
        word    overlay;        /* overlay number.                        */
}
EXE_HEADER;

static int      Hsize = 0 , Ssize = 0 ;

ARG     Argtab[] =
{
        { 'm' , INTEGER, &Hsize, "Set miminum heap size to <num>"  },
        { 's' , INTEGER, &Ssize, "Set stack size to <num>"         }
};

#define TSIZE   (sizeof(Argtab)/sizeof(ARG))

/*----------------------------------------------------------------------*/

usage()
{
        fprintf( stderr, "exe [-ms[<num>]] file\n" );
        exit(1);
}

/*----------------------------------------------------------------------*/

main( argc, argv )
char    **argv;
{
        EXE_HEADER      h;
        int             fd;
        unsigned        numpara, ostack, odata ;

        argc = getargs( argc, argv, Argtab, TSIZE, usage );

        if( argc != 2 )
                fatal_err("exe: exactly one file name required\n");


        if( (fd = open( argv[1], O_RDWR | O_BINARY )) == -1 )
                fatal_err( "Can't open %s\n", argv[1] );

        if( read( fd, (char *) &h, sizeof(h) ) != sizeof(h) )
                fatal_err( "Can't read %s\n", argv[1] );


        if( Hsize )
        {
             /*  1) numpara = the number of paragraphs required to hold the
              *              specified number of bytes.
              *  2) h.bss.max, the maximum heap size, gets either the
              *              current minimum or the specified size,
              *              whichever is larger.
              *  3) write out the modified header.
              */

             numpara = Hsize/16 + (Hsize % 16 != 0) ;                /* 1 */

             h.bss_max = (numpara<h.bss_min) ? h.bss_min : numpara;  /* 2 */

             lseek( fd, 0L, 0 );                                     /* 3 */
```

(continued on page 102)

Now You Know Why BRIEF™ is BEST

"BRIEF is simple to learn and use and extremely sophisticated."

PC Magazine, July 1986

COMPILER SUPPORT

No matter what compiler you have, it will run inside BRIEF. If errors occur during compilation, the supplied macros place your cursor on the line with the first problem and display the compiler's message. After you make your corrections, you skip to the next error with one keystroke. BRIEF automatically moves your cursor to the right place, even if you've added or deleted lines.

BRIEF is preconfigured (using the built-in macro language) for the Microsoft Macro Assembler v 4.0, and the Microsoft, Computer Innovations, Lattice, and Wizard C compilers. If you use another product, you can modify the macros to support it.

The Program Editor with the BEST Features

Since its introduction, BRIEF has been sweeping programmers off their feet. Why? Because BRIEF offers the features **MOST ASKED FOR** by professional programmers. In fact, BRIEF has just about every feature you've ever seen or imagined, including the ability to configure windows, keyboard assignments, and commands to **YOUR** preference. One reviewer (David Irwin, DATA BASED ADVISOR) put it most aptly, "(BRIEF)...is quite simply the best code editor I have seen."

Every Feature You Can Imagine

Compare these features with your editor (or any other for that matter).

- FAST
- Full UNDO (N Times)
- Edit Multiple Large Files
- Compiler-specific support, like auto indent, syntax check, compile within BRIEF, and template editing
- Exit to DOS inside BRIEF
- Uses all Available Memory
- Tutorial
- Repeat Keystroke Sequences
- 15 Minute Learning Time
- Windows (Tiled and Pop-up)
- Unlimited File Size -(even 2 Meg!)
- Reconfigurable Keyboard
- Context Sensitive Help
- Search for "regular expressions"
- Mnemonic Key Assignments
- Horizontal Scrolling
- Comprehensive Error Recovery
- A Complete Compiled Programmable and Readable Macro Language
- EGA and Large Display Support
- Adjustable line length up to 512

Program Editing YOUR Way

A typical program editor requires you to adjust your style of programming to its particular requirements - NOT SO WITH BRIEF. You can easily customize BRIEF to your way of doing things, making it a natural extension of your mind. For example, you can create ANY command and assign it to ANY key - even basic function keys such as cursor-control keys or the return key.

The Experts Agree

Reviewers at BYTE, INFOWORLD, DATA BASED ADVISOR, and DR. DOBB'S JOURNAL all came to the same conclusion - BRIEF IS BEST!

Further, of 20 top industry experts who were given BRIEF to test, 15 were so impressed they scrapped their existing editors!

NOT COPY PROTECTED

Solution Systems™

MONEY-BACK GUARANTEE
Try BRIEF ($195) for 30 days - If not satisfied get a full refund.
TO ORDER CALL (800-821-2492)

SOLUTION SYSTEMS, 335-D WASHINGTON ST., NORWELL, MA 02061, 617-659-1571

BRIEF is a trademark of UnderWare

Circle **no. 142** on reader service card.

Listing One *(Listing continued, text begins on page 104.)*

```
                write( fd, (char *) &h, sizeof(h) );
        }

        if( Ssize )
        {
                /* 1)   ostack  = number of paragraphs in original stack
                 * 2)   odata   = number of paragraphs of data.
                 * 2)   numpara = number of paragraphs in new stack.
                 * 4)   modify stack size.
                 * 5)   Adjust the size of the stack+data area as appropriate.
                 * 6)   write the modified header out to the file.
                 */

                ostack  = h.init_sp/16 + (h.init_sp % 16 != 0) ;        /* 1 */
                odata   = h.bss_max - ostack ;                          /* 2 */
                numpara = Ssize/16 + (Ssize % 16 != 0) ;                /* 3 */

                h.init_sp = Ssize ;                                     /* 4 */

                h.bss_min = odata + numpara;                            /* 5 */

                if( h.bss_min > h.bss_max )
                        h.bss_max = h.bss_min;

                lseek( fd, 0L, 0 );                                     /* 6 */
                write( fd, (char *) &h, sizeof(h) );
        }

        print_hdr( &h );
        close( fd );
}

/*----------------------------------------------------------------------*/

print_hdr( h )
EXE_HEADER      *h;
{
        printf("%6d (0x%04x): ", h->signature, h->signature );
        printf("Signature (marks this as a valid .exe file)\n");

        printf("%6d (0x%04x): ", h->image_len, h->image_len );
        printf("Length of image mod 512\n" );

        printf("%6d (0x%04x): ", h->file_size, h->file_size );
        printf("File size (512-byte blocks) including header\n");

        printf("%6d (0x%04x): ", h->num_reloc, h->num_reloc );
        printf("Number of relocation table entries\n");

        printf("%6u (0x%04x): ", h->header_size, h->header_size );
        printf("Size of the header (paragraphs) = %lu bytes\n",
                                (unsigned long) h->header_size * 16 );

        printf("%6u (0x%04x): ", h->bss_min, h->bss_min );
        printf("Min. memory above program (paragraphs) = %lu bytes\n",
                                (unsigned long) h->bss_min * 16 );

        printf("%6u (0x%04x): ", h->bss_max, h->bss_max );
        printf("Max. memory above program (paragraphs) = %lu bytes\n",
                                (unsigned long) h->bss_max * 16 );

        printf("%6d (0x%04x): ", h->stack_disp, h->stack_disp );
        printf("Displacement (paragraphs) of stack within load module\n");

        printf("%6d (0x%04x): ", h->init_sp, h->init_sp );
        printf("Initial value of the SP register (= the stack size)\n");

        printf("%6d (0x%04x): ", h->checksum, h->checksum );
        printf("Checksum for file\n");

        printf("%6d (0x%04x): ", h->init_ip, h->init_ip );
        printf("Initial value of the PC (IP) register\n");

        printf("%6d (0x%04x): ", h->code_disp, h->code_disp );
        printf("Displacement (paragraphs) of code seg. within load module\n");

        printf("%6d (0x%04x): ", h->first_reloc, h->first_reloc );
        printf("Displacement (bytes) to first relocation item in module\n");

        printf("%6d (0x%04x): ", h->overlay, h->overlay );
        printf("Overlay number.\n");
}
```

End Listing One

Listing Two

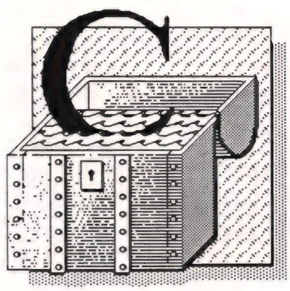
```
#include <stdio.h>
#include <stdarg.h>

fatal_err( fmt )                /* Print an error message to stderr and  */
char    *fmt;                   /* then exit.                            */
{
        va_list   args;
        va_start( args, fmt );
        vfprintf( stderr, fmt, args );
        exit( 1 );
}
```

End Listing Two

Listing Three

```
#include <stdio.h>                     /*  NRTRAV.C: A non-recursive binary */
                                       /*  tree traversal routine that uses */
typedef struct _n                      /*  the link-inversion method.       */
{
        int         tag;
        struct _n   *left;
        struct _n   *right;
        char        *key;
}
NODE;

/*-------------------------------------------------------------------------*/

#define print(nodep)    printf( "%s ", (nodep)->key );

/*-------------------------------------------------------------------------*/

descend_left( pres, prev )              /* Descend left till we can't   */
NODE    **pres, **prev;                 /* go any farther, reversing    */
{                                       /* links.                       */
        register NODE   *next;

        while( next = (*pres)->left )
        {
                (*pres)->left = *prev;
                *prev         = *pres;
                *pres         = next;
        }
}

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/

descend_right( pres, prev )             /* Descend right one node,      */
NODE    **pres, **prev ;                /* reversing links.             */
{                                       /* Return 0 if we couldn't go.  */
        register NODE   *next;

        if( !(next = (*pres)->right) )
                return 0;

        (*pres)->tag   = 1;
        (*pres)->right = *prev;
        *prev          = (*pres);
        *pres          = next;
        return 1;
}

/* - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/

trav( pres )
NODE  *pres;
{
        NODE    *prev = NULL, *next;

        do
        {
                descend_left( &pres, &prev );
                print( pres );
        } while( descend_right( &pres, &prev) );

        while( prev )
        {
                if( prev->tag == 0 )                  /* go up from a */
                {                                     /* left child   */
                        next       = prev->left;
                        prev->left = pres;
                        pres       = prev;
                        prev       = next;

                        while( 1 )                    /* and back down */
                        {
                                print( pres );
                                if( !descend_right(&pres, &prev) )
                                        break;

                                descend_left( &pres, &prev );
                        }
                }
                else                                  /* go up from a   */
                {                                     /* right child    */
                        next        = prev->right;
                        prev->tag   = 0;
                        prev->right = pres;
                        pres        = prev;
                        prev        = next;
                }
        }
}
```

End Listings



Shrinking .EXE File Images

It's been pointed out to me that the shell occupies much more memory at run time than is actually needed. Fortunately, the problem is easy to fix without having to recompile, and the techniques used are applicable to all .EXE files.

The problem has to do with how .EXE files are loaded into memory by MS-DOS and with how memory is used by *malloc( )* and *free( )*. (See this month's Flotsam and Jetsam, page 108, for a description of memory organization within a C program.)

The MS-DOS loader reads the text and data segments from the disk and then allocates all remaining memory for the bss segment, stack, and heap, even if the program is a small-model program that couldn't possibly use all that memory. When an .EXE file is loaded, DOS can reduce this default to an amount of memory specified in a header found at the beginning of the file (in the first 14 words). This amount has to be large enough to accommodate the entire bss and stack segments. The heap, however, doesn't need to be allocated at load time because more memory is requested from DOS if the heap isn't large enough when *malloc( )* is called, thereby increasing the size of the run-time image.

The .EXE file header is initialized by the linker so that all available memory is allocated to the current program. Most small-model programs will then reduce this amount to a 64K

by Allen Holub

combined data/bss/stack/heap area as part of the boot process. Unfortunately, 64K is always allocated, whether or not you need it. The shell, in its released form, has this 64K data space allocated to it, even though it only needs about 3K for static data and another 3K for the heap. Consequently it takes up almost 90K of memory rather than the 50K or so that's actually needed.

The automatic assignment of 64K can be circumvented by having the linker put a more reasonable number into the .EXE file header (by using the */CP* or */STACK* command-line options). It's not always convenient to relink, however, especially if you don't have the original source or object modules. Fortunately, the size of the run-time image can be reduced by doing nothing more than changing a couple of numbers in the .EXE file header.

The Microsoft C compiler comes with a nifty little program called exemod that does just that—messes around with the .EXE file header to change the default run-time size of the program. Unfortunately the Microsoft version is needlessly difficult to use (requiring you to specify stack sizes in hex bytes and heap sizes in decimal paragraphs), and, of course, if you don't have the compiler, you don't have exemod either. An easier-to-use version of exemod is in Listing One (page 100).

The program (called exe) can be used in one of three ways (shown in Table 1, page 107, along with a sample output). If no command-line switches are present, then exe just prints the contents of the header. I'll look at this header in greater depth in a moment. The *−m* flag is used to change the default data area size (the combined sizes of the stack, heap, and bss areas). If N is too small (less than the combined bss and stack sizes), then it's rounded up to the minimum. You can use *−m1* to get the smallest possible run-time image, though the image will grow larger if the program ever calls *malloc( )*. The *−sN* switch increases or decreases the stack size to N bytes. If necessary, the run-time image will be made larger to accommodate a larger stack. The image isn't made smaller when you reduce the stack size. You can run exe twice, however, reducing the stack size the first time and then reducing the total file size the second time. For example:

```
exe −s1024 file.exe
exe −m1 file.exe
```

reduces a file's stack to 1,024 bytes and then eliminates the space allocated to the heap. Be careful about reducing the stack of a Microsoft-compiled program to less than 1K—I always seem to get a stack overflow error message when I do this. God knows what all that stack space is used for—my own part of the program isn't using it.

Note that the largest N that can be associated with either switch is 65,535. The only reason for this limitation is that I've used *getargs( )* to process command-line arguments and *getargs( )* can't handle *long*-size arguments very easily. If you want larger images, replace the *getargs( )* call on line 68 with your own command-line processing routine.

The .EXE file header is defined by the structure on lines 22−39, reproduced in Code Example 1, page 107. The *signature* is a unique number used to identify this file as an .EXE file. The *file_size*, *header_size*, and *image_len* fields are used to determine the size of the load module (the combined text and initialized data areas). In particular, the load image requires

```
((file_size * 512) − (header_size * 16))
                            + image_len
```

Technical Product Information.

Dr. Dobb's Journal of Software Tools

Name ______________________________

Title ______________________________

Company ____________________ Phone ____________________

Address ______________________________

City/State/Zip ______________________________

January 1987 #123 **Expiration Date:** **April 30, 1987**

Please circle one letter in each category:

I. My work is performed:
A. for in-house use only.
B. for other companies.
C. for end users/retailers.
D. in none of the above areas.

II. My primary job function:
A. Software Project Mgmt/Spvr
B. Hardware Project Mgmt/Spvr
C. Computer Consultant
D. Corporate Consultant
E. Other

III. My company department performs:
A. software development.
B. computer system integration.
C. computer manufacturing.
D. computer consulting.
E. computer research
F. none of the above.

IV. This inquiry is for:
A. a purchase within 1 month.
B. a purchase within 1 to 6 months.
C. product information only.

V. Corporate Purchase Authority:
A. Final Decision-maker
B. Approve/Recommend
C. No Influence

VI. Personal Computer Users at my Jobsite:
A. 10,000 or more
B. 500 to 9,999
C. 100 to 499
D. 10 to 99
E. less than 10

VII. On average, I advise others about computers:
A. more than once per day.
B. once per day.
C. once per week.
D. less than once per week.

VIII. In my job function, I:
A. design software and/or write code.
B. design software.
C. write code.
D. don't design software or write code.

A Reader Service number appears on each advertisement. Circle the corresponding numbers at right for more info.

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120
121	122	123	124	125	126	127	128	129	130
131	132	133	134	135	136	137	138	139	140
141	142	143	144	145	146	147	148	149	150
151	152	153	154	155	156	157	158	159	160
161	162	163	164	165	166	167	168	169	170
171	172	173	174	175	176	177	178	179	180
181	182	183	184	185	186	187	188	189	190
191	192	193	194	195	196	197	198	199	200
201	202	203	204	205	206	207	208	209	210
211	212	213	214	215	216	217	218	219	220
221	222	223	224	225	226	227	228	229	230
231	232	233	234	235	236	237	238	239	240
241	242	243	244	245	246	247	248	249	250
251	252	253	254	255	256	257	258	259	260
261	262	263	264	265	266	267	268	269	270
271	272	273	274	275	276	277	278	279	280
281	282	283	284	285	286	287	288	289	290
291	292	293	294	295	296	297	298	299	300
301	302	303	304	305	306	307	308	309	310
311	312	313	314	315	316	317	318	319	320
321	322	323	324	325	326	327	328	329	330
331	332	333	334	335	336	337	338	339	340
341	342	343	344	345	346	347	348	349	350
351	352	353	354	355	356	357	358	359	360
361	362	363	364	365	366	367	368	369	370
371	372	373	374	375	376	377	378	379	380
381	382	383	384	385	386	387	388	389	390
391	392	393	394	395	396	397	398	399	999

Circle 999 to start a 12 month subscription at the price of $29.97

TAKE THIS CARD WITH YOU AS YOU READ THROUGH THIS ISSUE OF DR. DOBB'S.

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL
First Class Permit #217, Clinton, Iowa

POSTAGE WILL BE PAID BY ADDRESSEE

Dr. Dobb's Journal of
Software Tools
FOR THE PROFESSIONAL PROGRAMMER

P.O. Box 2157
Clinton, Iowa 52735-2157

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
FIRST CLASS PERMIT #217, CLINTON, IOWA

POSTAGE WILL BE PAID BY ADDRESSEE

Dr. Dobb's Journal of
Software Tools
FOR THE PROFESSIONAL PROGRAMMER

P.O. Box 2157
Clinton, Iowa 52735-2157

Technical Product Information.

TAKE THIS CARD WITH YOU AS YOU READ THROUGH THIS ISSUE OF DR. DOBB'S.

1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60
61	62	63	64	65	66	67	68	69	70
71	72	73	74	75	76	77	78	79	80
81	82	83	84	85	86	87	88	89	90
91	92	93	94	95	96	97	98	99	100
101	102	103	104	105	106	107	108	109	110
111	112	113	114	115	116	117	118	119	120
121	122	123	124	125	126	127	128	129	130
131	132	133	134	135	136	137	138	139	140
141	142	143	144	145	146	147	148	149	150
151	152	153	154	155	156	157	158	159	160
161	162	163	164	165	166	167	168	169	170
171	172	173	174	175	176	177	178	179	180
181	182	183	184	185	186	187	188	189	190
191	192	193	194	195	196	197	198	199	200
201	202	203	204	205	206	207	208	209	210
211	212	213	214	215	216	217	218	219	220
221	222	223	224	225	226	227	228	229	230
231	232	233	234	235	236	237	238	239	240
241	242	243	244	245	246	247	248	249	250
251	252	253	254	255	256	257	258	259	260
261	262	263	264	265	266	267	268	269	270
271	272	273	274	275	276	277	278	279	280
281	282	283	284	285	286	287	288	289	290
291	292	293	294	295	296	297	298	299	300
301	302	303	304	305	306	307	308	309	310
311	312	313	314	315	316	317	318	319	320
321	322	323	324	325	326	327	328	329	330
331	332	333	334	335	336	337	338	339	340
341	342	343	344	345	346	347	348	349	350
351	352	353	354	355	356	357	358	359	360
361	362	363	364	365	366	367	368	369	370
371	372	373	374	375	376	377	378	379	380
381	382	383	384	385	386	387	388	389	390
391	392	393	394	395	396	397	398	399	999

Circle 999 to start a 12 month subscription at the price of $29.97

Dr. Dobb's Journal of Software Tools

Name ______________________
Title ______________________
Company ____________ Phone ____________
Address ______________________
City/State/Zip ______________________

January 1987 #123 **Expiration Date:** **April 30, 1987**

Please circle one letter in each category:

I. My work is performed:
A. for in-house use only.
B. for other companies.
C. for end users/retailers.
D. in none of the above areas.

II. My primary job function:
A. Software Project Mgmt/Spvr
B. Hardware Project Mgmt/Spvr
C. Computer Consultant
D. Corporate Consultant
E. Other

III. My company department performs:
A. software development.
B. computer system integration.
C. computer manufacturing.
D. computer consulting.
E. computer research
F. none of the above.

IV. This inquiry is for:
A. a purchase within 1 month.
B. a purchase within 1 to 6 months.
C. product information only.

V. Corporate Purchase Authority:
A. Final Decision-maker
B. Approve/Recommend
C. No Influence

VI. Personal Computer Users at my Jobsite:
A. 10,000 or more
B. 500 to 9,999
C. 100 to 499
D. 10 to 99
E. less than 10

VII. On average, I advise others about computers:
A. more than once per day.
B. once per day.
C. once per week.
D. less than once per week.

VIII. In my job function, I:
A. design software and/or write code.
B. design software.
C. write code.
D. don't design software or write code.

A Reader Service number appears on each advertisement. Circle the corresponding numbers at left for more info.

C Programmers! High-Speed Database tames complex C applications

"db_VISTA™ lets you easily build complex databases with many interconnected record types..."
Dave Schmitt, President, Lattice, Inc.

High-Speed data retrieval and access... just two benefits of using RAIMA's network model DBMS, db_VISTA. Combine these design benefits with those of C—speed, portability, efficiency, and you begin to understand db_VISTA's real measure... performance.

Independent Benchmark proves High-Speed model 2.76 times faster

An independent developer benchmarked db_VISTA against a leading competitor. Eleven key retrieval tests were executed with sequentially and randomly created key files.

*Result of 11 Key Retrieval Tests

db_VISTA	:671.24 seconds
Leading Competitor	:1,856.43 seconds

db_VISTA's high-speed network database model lets you precisely define relationships to minimize redundant data. Only those functions necessary for operation are incorporated into the run time program.

Portable DBMS Applications with db_VISTA

For maximum application portability, every line of db_VISTA's code is written in C and complete source code is available. db_VISTA operates on most popular computers with several operating systems supported. So whether you write applications for micros, minis, or mainframes...db_VISTA is for you.

How db_VISTA works...

Design your database and compile your schema file with the database definition language processor. Develop application programs, making calls to db_VISTA's C functions. Edit and review your database using the Interactive Database Access utility. Compile and link your C program with the db_VISTA runtime library, and your application is ready to run.

Multi-user and LAN capability

Information often needs to be shared. db_VISTA has multi-user capability and supports simultaneous users in either multi-tasking or local area networking environments, allowing the same C applications to run under UNIX, MS-DOS, and VAX VMS.

db_QUERY™ lets you ask more of your database

db_QUERY is a linkable, royalty-free, SQL-based ad hoc query and report writing facility. It provides a user-friendly relational view of a network-model database. Use it directly or design an interface for the inexperienced to generate powerful queries and reports.

Royalty-Free Run-Time

Whether you're developing applications for a few customers, or for thousands, the price of db_VISTA or db_QUERY is the same. If you are currently paying royalties for a competitor's database, consider switching to db_VISTA and say goodbye to royalties.

FREE Technical Support For 60 days

Raima's software includes free telephone support and software updates for 60 days. Technical support personnel are available to answer questions about our software or yours.

30-Day Money-Back Guarantee

Try db_VISTA for 30 days and if not fully satisfied, return it for a full refund.

Order Schedule

	db_VISTA	db_QUERY
☐ Single-user	$ 195	$ 195
☐ Single-user w/Source	$ 495	$ 495
☐ Multi-user	$ 495	$ 495
☐ Multi-user w/Source	$ 990	$ 990
☐ VAX Multi-user	$ 990	$ 990
☐ VAX Multi-user w/Source	$1980	$1980

Not Copy Protected

Call Toll-Free Today!

Order Line 1-800-327-2462
Information Line . 1-206-828-4636

Read what others say...

"If you are looking for a sophisticated C programmer's database, db_VISTA is it. In either a single or multi-user environment, db_VISTA lets you easily build complex databases with many interconnected record types. The multi-user implementation handles data efficiently with a LAN, and Raima's customer support and documentation are excellent. Source code availability and a royalty-free run-time is a big plus."
Dave Schmitt, President Lattice, Inc.

"My team has developed a sophisticated PC-based electronic mail application for resale to HP customers. db_VISTA has proved to be an all-round high performer in terms of fast execution, flexibility and portability, and has undoubtedly saved us much time and development effort."
John Adelus, Hewlett-Packard Ltd. Office Productivity Division

"On the whole, I have found db_VISTA easy to use, very fast with a key find, and powerful enough for any DBMS use I can imagine on a microcomputer."
Michael Wilson, Computer Language

db_VISTA Version 2.2

Database Record and File Sizes
- Maximum record length limited only by accessible RAM
- Maximum records per file is 16,777,215
- No limit on number of records or set types
- Maximum file size limited only by available disk storage
- Maximum of 255 index and data files

Keys and Sets
- Key length maximum 246 bytes
- No limit on maximum number of key fields per record—any or all fields may be keys with the option of making each key unique or duplicate
- No limit on maximum number of fields per record, sets per database, or sort fields per set
- No limit on maximum number of member record types per set

Operating System & Compiler Support
- Operating systems MS-DOS, PC-DOS, UNIX, XENIX, SCO XENIX, UNOS, ULTRIX, VMS
- C compilers: Lattice, Microsoft, DeSmet, Aztec, Computer Innovations, XENIX and UNIX

Features
- Multi-user support allows flexibility to run on local area networks
- File structure is based on the B-tree indexing method and the network database model
- Run-time size, variable—will run in as little as 64K, recommended RAM size is 256K
- Transaction processing assures multi-user database consistency
- File locking support provides read and write locks on shared databases
- SQL-based db_QUERY is linkable
- File transfer utilities included for ASCII, dBASE optional

Utilities
- Database definition language processor
- Interactive database access utility
- Database consistency check utility
- Database initialization utility
- Multi-user file locks clear utility
- Key file build utility
- Data field alignment check utility
- Database dictionary print utility
- Key file dump utility
- ASCII file import and export utility

*The benchmark procedure was adapted from "Benchmarking Database Systems: A Systematic Approach" by Bitton, DeWitt and Turbyfill, December 1983.

RAIMA™ CORPORATION

High-Speed Programming Tools, Designed for Portability

3055-112th Avenue N.E.
Bellevue, WA 98004 USA
(206) 828-4636 Telex: 9103330300

Order Toll-Free
1 (800) 327-2462

C CHEST
(continued from page 104)

bytes. In Table 1, this comes to

((22 * 512)−(32 * 16)) + 124

or 10,876 bytes.

Several of the fields are used for patching up a few instructions that the linker can't patch. There are *num_reloc* of these items (three in exe.exe) organized as a linked list with the first node in the list at offset *first_reloc* from the beginning of the load module.[1]

The *bss_min* and *bss_max* fields are used to allocate the combined heap, stack, and bss space. The initialized data, because it's stored on the disk, is considered to be part of the load module, so its size isn't duplicated here. *Bss_min* is the minimum amount of required memory in paragraphs (16-byte chunks). It's the combined bss and stack sizes. *Bss_max* determines the maximum amount of allocated memory (also in paragraphs), so if it's larger than *bss_min*, the difference between the two numbers is the amount of heap that can be allocated before DOS has to be called. The default values of *bss_min* and *bss_max* for exe.exe are shown in Table 1 (196 and 65,535, respectively). This means that the program requires a minimum of 196 paragraphs (3,136 bytes) for the combined bss/stack area and will use all the rest of memory for the heap. The smallest-possible image can be created by setting *bss_max* to *bss_min*.

The *init_sp* and *stack_disp* fields are used to set up the stack size. *Init_sp* is both the stack size and the initial value of the *SP* register (the *SS* register points at the bottom of the stack area). *Stack_disp* is used to locate the bottom of the stack. It is added to the initial contents of the *CS* register to initialize the *SS* register when DOS loads the program. Note that the stack size is included in the *bss_min* and *bss_max* figures, so these will have to be modified if *init_sp* is made larger.

All these transformations are done by the code in Listing One. The file is opened on line 74, the .EXE header is read on line 77, the *bss_min* and *bss_max* fields are modified on lines 81−97, and the stack variables are modified on lines 99−122. The .EXE header is written back out on both lines 95−96 and 120−121 (you have to seek back to the start of the file before writing). Finally, the header contents are printed by *print_hdr( )*, called on line 124.

The *fatal_err( )* subroutine is given in Listing Two, page 102. It is used just like *printf( )* is used. It writes a message to *stderr* and then exits to the operating system. Note that I've used the ANSI (as compared to Unix) conventions for subroutines with a variable number of arguments. *Va_list* and *va_start* are macros defined in stdarg.h, supplied with the compiler. If your compiler doesn't support these, substitute calls to *fprintf( )* and then *exit( )* for the *fatal_err( )* calls. I'll talk more about subroutines with a variable number of arguments in a future column.

Erratum

The nonrecursive binary-tree traversal routine presented in July has a serious bug in the algorithm. It couldn't handle the case of a leaf that had a right, but no left, descendant. Listing Three, page 103, is another version of the routine that seems to work correctly. The basic process is

still the same (descend the tree reversing pointers so you can go back up again, setting a tag bit just before going right), but the code has been shuffled around a bit. Note that in this version I'm keeping a *tag* field in the structure rather than setting the high bit of the first character of the key string. Look back in the July C Chest if you need a more detailed explanation of what's going on.

Availability

All the code from this month is available on CompuServe in DL1 (type ddjforum). The *getargs( )* subroutine, used but not printed this month, was originally published in the May 1985 C Chest. The version used here has a fifth argument not present in the original version. If you're using the earlier version, just omit the extra argument. The current version of *getargs* is available both on CompuServe and as part of the /util program disk distributed by *DDJ* (see ad, page 124).

All the source code for articles in this issue is available on a single disk. To order, send $14.95 to *Dr. Dobb's*

exe −mN file	Modify the maximum memory used to N bytes. If this number is smaller than the combined bss and stack sizes, then it's rounded up. Use *−m1* for the smallest possible load module. The maximum value of N is 65,535.
exe −sN file	Modify the stack size to be N bytes.
exe file	Just print out the contents of the .EXE header. The command line *exe exe.exe* generated the following:

23117 (0x5a4d):	Signature (marks this as a valid .EXE file)
124 (0x007c):	Length of image mod 512
22 (0x0016):	File size (512-byte blocks) including header
3 (0x0003):	Number of relocation table entries
32 (0x0020):	Size of the header (paragraphs) = 512 bytes
196 (0x00c4):	Min. memory above program (paragraphs) = 3,136 bytes
65535 (0xffff):	Max. memory above program (paragraphs) = 1,048,560 bytes
715 (0x02cb):	Displacement (paragraphs) of stack within load module
2048 (0x0800):	Initial value of the *SP* register (= the stack size)
−14653 (0xc6c3):	Checksum for file
2008 (0x07d8):	Initial value of the *PC* (*IP*) register
0 (0x0000):	Displacement (paragraphs) of code seg. within load module
30 (0x001e):	Displacement (bytes) to first relocation item in module
0 (0x0000):	Overlay number

Table 1: *Using exe*

```
typedef unsigned short word;  /* 2-byte unsigned number */

typedef struct
{
    word   signature;
    word   image_len;     /* Length of load module image % 512      */
    word   file_size;     /* File size in 512-byte units            */
    word   num_reloc;     /* Number of relocation table items       */
    word   header_size;   /* Size of the header in paragraphs       */
    word   bss_min;       /* min size of data area above program    */
    word   bss_max;       /* max size of data area above program    */
    word   stack_disp;    /* displacement in para. of stack seg.    */
    word   init_sp;       /* Initial SP register contents           */
    word   checksum;      /* Checksum for file                      */
    word   init_ip;       /* Initial IP register contents (PC)      */
    word   code_disp;     /* displacement in para. to code seg.     */
    word   first_reloc;   /* displacement (bytes) to 1st re-
                             loc item                               */
    word   overlay;       /* overlay number.                        */
}
EXE_HEADER;
```

Code Example 1: *The .EXE file header*

C CHEST
(continued from page 107)

Journal, 501 Galveston Dr., Redwood City, CA 94063 or call (415) 366-3600 ext. 216. Please specify the issue number and disk format (MS-DOS, Macintosh, Kaypro).

Note

1. For more information about how the relocations are processed consult Chapter 10 of IBM Corp.'s *DOS Technical Reference* (Boca Raton, Fla.: IBM Corp., 1985). Much better descriptions of relocatable object module formats in general are in Steven Armbrust and Ted Forgeron's ".OBJ Lessons," *PC Tech Journal* 3:10 (October 1985), 63–81, and Intel Corp.'s *8086 Relocatable Object Module Formats* (Santa Clara, Calif.: Intel Corp., 1981), order number 121748-001.

DDJ

(Listings begin on page 100.)

Vote for your favorite feature/article.
Circle Reader Service **No. 6.**

Flotsam and Jetsam

Memory Organization in a C Program

Most C programs are segmented into five parts: the code area (called the *text* segment); the initialized data space (called the *data* segment); the uninitialized data space (or *bss* segment; the stack space (or *stack* segment); and the area of memory used by *malloc( )*, called the *heap*.

The stack is used for subroutine calling, in the normal way, but it's also used to store local automatic variables. When a subroutine is called, it subtracts a constant from the stack pointer to make room on the stack for its own local variables and then accesses those variables indirectly through either the stack pointer or a special register called the *frame pointer*.

Variables at fixed addresses (globals and local statics) are in either the data or bss segments, depending on whether they are initialized by your program. Variables can be initialized explicitly (with an equal sign as part of the declaration) or implicitly. An example of the latter is a string constant, such as a format string passed to a *printf( )* call. Here the compiler automatically allocates and implicitly initializes an area of memory to hold the string, and that memory is put into the data segment (rather than the bss segment).

Usually, both the text and data segments (code and initialized data areas) are stored on the disk together. When the program is loaded, the variables in the data segment are loaded directly into their correct place in memory. There's no code generated to initialize static data; the data that is read in from the disk has the correct initial value. This explains why a static local variable has its initial value the first time a subroutine is called but on subsequent calls the variable contains the same value that it had at the end of the previous call. It's read in having the initial value, but once you change it, it stays changed.

The remaining three memory areas (bss, stack, and heap) are created as part of the loading process. After the loader has transferred the text and data segments from the disk into memory, it allocates enough additional memory above the data segment to contain the other three segments. The loader usually sets up the stack pointer to point into the stack. The program itself (or more correctly, the *root* or *start-up* module) initializes the entire bss segment with 0s and then calls your *main( )* subroutine.

The usual order of segments, going from low to high memory, is:

text	(code)
data	(initialized data)
bss	(uninitialized data)
stack	(local variables)
heap	(used by *malloc( )*)

However, the stack and heap are often reversed. The text and data areas are always adjacent because they're read from the disk as a single unit.

Naming Names

Forth has been designed by programmers who were using it, and so Forth's design is responsive to programmers' needs in small ways as well as large. Other languages don't seem to be quite so programmer-friendly. For example, I was surprised to read in an article about Modula-2 (by this column's own Namir Shammas) the complaint that the underscore character was not allowed in names. How did Modula-2's designer conclude that programmers are helped by disallowing some characters in names? In my heart of hearts, I suspect that the rule was for the benefit of the compiler writers, not the compiler users, and exemplifies fitting the task to the program rather than the other way around. This type of programming focuses on what is easy now (for the program writer), not what is easy over the life of the program (for the program users).

Service workers must always fight the tilt toward serving themselves before the clients of their profession. College administrators who bemoan the loss of serenity when students return to the campus, shelf stockers whose tempers flare when customers disorganize displays by buying items from them, and programmers who have had it up to here with figuring how to help the endlessly confused user—all should remind themselves of the point of the enterprise.

Programmers using a particular language pray that its developers kept in mind that they were writing for programmers and made their first objective easing the programmer's life, not their own task. The programmer users hope that the language developers, weary of considering all the ins and outs of implementation, did not finally throw in the towel and say, "This will be good for you, really. You'll like not being able to use underscores. Anyway, you'll get used to it."

by Michael Ham

In Forth, any character can be used in a name—well, almost any. Blank doesn't work because it is the name delimiter, and carriage return doesn't work because it marks the end of the line. Generally speaking, however, Forth is not picky about the characters you want to use.

Some standard usages have evolved in which some characters represent a class of tasks. A word beginning with a period normally displays information: *.DATE*, for example, can be assumed to display the date, *.NAME* a name, *.S* the stack (abbreviated because so often used), *.FILENAME* a file name, and so on. The greater-than symbol is often used for "to"—to indicate movement (*>R* puts a character from the data stack onto the return stack, *R>* takes it off the return stack and returns it to the data stack) or transformation (*S>D* converts a single-precision number to a double-precision equivalent, *>JULIAN* converts a date to a Julian date). The symbol *?* denotes a Boolean flag, and in a program in which names are carefully chosen, its meaning is obvious. For example, *STOP?* would leave a flag true, meaning stop, and *?STOP* would consume a flag true, causing a stop.

Code Example 1, below, shows a tiny tool *YES?* that collects a yes/anything response and leaves a true flag if the user answers yes. The word suggests a yes response (hence the name) by displaying *Y* as the default answer. *YES?* uses *CAP* to capitalize any lowercase input before checking whether it was a *Y*.

Some of these name patterns come from conventions, but conventions are more successful when they recognize and reinforce usage than when they attempt to create it. Rushing the process or trying to fence it in with rules does not lead to better results more quickly but merely frustrates and confuses the evolutionary movement. Forth gives the programmer complete freedom in naming, and the conventions for naming emerge gradually.

Other languages give the compiler writer the authority to decide the sorts of names that would be good for programmers (or, possibly, good for compiler writers) with the result that some characters fall beyond the pale that pens the programmer. "You want underscores in the name? That sort of thing isn't done in Modula-2. Don't be perverse."—the mark of bluestockings, ready to ease their life by adding difficulties to yours. Fight back. Use Forth.

Some programming languages build in conventions through tactics such as precedence rules. One popu-

```
: CAP ( c — C ) DUP 96 > OVER 123 < AND IF BL - THEN ;

: YES? ( — f ) ASCII Y EMIT 8 EMIT ( backspace )
   KEY CAP DUP ASCII Y = SWAP 13 = ( cr? ) OR DUP
      IF ASCII Y ELSE ASCII N THEN EMIT SPACE ;
```

Code Example 1: *Two tiny tools*

lar language has upward of 20 precedence rules. It's too many. The doctor in John Barth's novel *End of the Road* (New York: Avon Books, 1964) suggests to Jacob Horner that three will suffice: " 'If the alternatives are side by side, choose the one on the left; if they're consecutive in time, choose the earlier. If neither of these applies, choose the alternative whose name begins with the earlier letter of the alphabet. These are the principles of Sinistrality, Antecedence, and Alphabetical Priority.' "

Of course, it is always legitimate to propose rules. Ideas can be stimulated through discussion, but their acceptance should ultimately be based upon experience, not fiat. To demonstrate that I am willing to entertain rule proposals, I offer the following suggestions for naming conventions for the arithmetic operators.

The names of the arithmetic operators are perhaps inescapably pedestrian. Forth requires a variety of names because data are not typed and thus the operators are. Operators come in several flavors: single precision, double precision, quad precision, and mixed precision (operations in which the two operands are of different precision, typically one being single precision and the other double precision). Happily, all the mixed-precision operators can be defined to take the lesser precision operand on top of the stack, the greater precision second on the stack.

To simplify the discussion, let's follow FORTH Inc.'s lead and call single-precision numbers *singles* and double-precision numbers *doubles*. There are no mixed-precision numbers, of course, only mixed-precision operations (with a single and a double or a double and a quad as arguments).

The precision of the result of an operation is another question. Normally sums and differences are assumed to have the same precision as the operands that produced them, though that is not logically necessary: a sum of two singles could, for example, be a double. In multiplication you more commonly will want to allow the result to be of a higher precision than the factors (the product of two singles being a double, for example). And with division you might be content for the quotient to drop back a notch: double divided by single with the quotient a single. Note, however, that it seems best for the remainder to be accepted as a double even if the quotient is taken as a single.

The sign lurks as the high bit of the binary representation of the number, but unsigned numbers use that bit in its numeric meaning. Addition and subtraction do not need to distinguish—the programmer can choose how to interpret the high bit when the result is displayed. But in other operations it can make a difference: is it 10 compared to 65,535 (10 is less) or 10 compared to −1 (10 is greater)? If an operation treats the high bit as number rather than as sign, the operation is called unsigned.

Ideally, the Forth names for the operators could offer the programmer some reliable signposts through this maze of options: single, mixed, double, unsigned, signed, incoming, outgoing. The current crop of names was not designed to offer this kind of

(continued from page 111)

help. An alternative scheme is suggested in Tables 1 and 2, below.

Table 1 contains a list of prefixes for the arithmetic operators, based on the precision of the operands. When both operands are signed, single-precision numbers, the operators are unadorned. Otherwise, the operator names include information that describes the nature of the operands.

Table 2 contains a list of suffixes. Just as the prefix describes the nature of the operands (the input), the suffix describes the nature of the result (the output). Again, a garden-variety operator that produces the natural result (for example, an operation on single-precision numbers that produces a single-precision result or on doubles that produces a double). This convention assumes that the "natural" result for a mixed-precision operation has the higher of the precisions of the two operands. For example, the natural result of a single-double mixed operator is a double, and thus no suffix is used in that case. If a single-double operator produces a single result, then it is named with a suffix *S* to show that the result is single precision.

Operands	Sign Assumption	Prefix
Single-precision	signed operands	none
	unsigned operands	U
Double-precision	signed operands	D
	unsigned operands	DU
Quad-precision	signed operands	Q
	unsigned operands	QU
Mixed-precision		
Single-double	signed operands	M
	unsigned operands	MU
Double-quad	signed operands	MD
	unsigned operands	MDU

***Table 1:** Prefixes for arithmetic operators*

Operands	Result	Suffix
Single-precision	single precision	none
	double precision	D
Double-precision	single precision	S
	double precision	none
	quad precision	Q
Mixed-precision		
Single-double	single precision	S
	double precision	none
Double-quad	double precision	D
	quad precision	none

***Table 2:** Suffixes for arithmetic operators*

*D	Two signed single-precision factors producing a double product.
U*D	Two unsigned single-precision factors producing a double product.
MU*	Mixed-precision factors (single and double), unsigned, with double product. By convention, the single-precision factor is on top of the stack, the double under it.
MDU*	Mixed-precision factors (double and quad), unsigned, with quad product.
MU>	Mixed-precision unsigned compare. The sense of the comparison is double > single, both considered as unsigned.
D*/	Factors and result are all doubles, with quad-precision intermediate product (The */ operator always takes the intermediate product to the next higher level of precision. Because operators in the */ family have three operands, it seems best to avoid mixed precision on the input side.)
D*	Factors and product all signed doubles.
M*	Single on top of stack, double beneath, with double-precision product, all signed.
D/	Two signed doubles divided, producing double as quotient.
D/S	Two signed doubles divided, producing signed single as quotient.
M/S	Double divided by single with single quotient, all signed.
D/MOD	Two doubles divided, producing a double quotient and a double remainder.

***Table 3:** Examples of operator names*

These names group double-precision operators under *D* and mixed-precision operators under *M*. The affixes allow you to decipher the special qualities of an arbitrary operator and also make it easy to remember the operation names. Table 3 shows a variety of operator names that test this scheme.

Mixed precision normally is an issue only with the input because the output is normally only one number. The */MOD* operators, however, produce two numbers as output: a quotient and a remainder. Would it be reasonable to see mixed precision here, with the quotient being one precision and the remainder another?

In integer division, the max of the dividend and the divisor will be larger than the quotient and the remainder, so if the dividend and divisor are both single precision, both quotient and remainder will be single precision. The single-precision */MOD* thus leaves single-precision results, and the issue does not arise.

With *D/MOD*, however, the situation is different. Dividing a double by a double might well produce a single. The remainder, however, could well be a double. Thus, you might want the operation *D/MODM*, two doubles producing a mixed result: a single-precision quotient and a double-precision remainder. Mirabile dictu, the single is again on top of the stack, the double beneath. (Perhaps single numbers, weighing less than doubles, naturally float to the top of the stack.) You can use *M* as a suffix to indicate that the mixed precision is on the output side rather than the input:

D/MODM*—three signed doubles, with quad-precision intermediate re-

sult, producing a mixed-precision result: single quotient on top of stack and double remainder beneath.

Q/MODM*—three unsigned quads, with octuple-precision intermediate result, producing a quad remainder (second on the stack) and a double quotient (top of stack).

The example *Q*/MODM* is a grotesquerie that you would probably never encounter. It serves merely to illustrate that even novel operations can be deciphered easily with this scheme.

If you accept that all *M* operators require the single number on top of the stack and the double beneath, you can define the following operators that might be useful in mixed-precision situations. But beware of the syndrome of maniacal completeness, in which you define fistfuls of operators to complete a set of logical possibilities, even if those operators are seldom or never used. I suggest that these words be defined only in applications doing a lot of mixed-precision calculation. In that setting, their descriptive names make the code more readable and thus justify their existence.

```
MSWAP  ( d n — n d )
MOVER  ( d n — d n d )
MNIP   ( d n — n )
MTUCK  ( d n — n d n )
```

Some of the names I have suggested for the arithmetic operators don't match names in the 83 Standard. I see no problem in this; the names I propose could be adopted and the older names kept as synonyms. Forth programmers can be encouraged to shift to the new names by having a tiny speed penalty associated with the older names—for example, by defining the older name as an alias of the newer name. Any speed penalty, however slight, is more than enough to make most Forth programmers switch to the faster name.

The Toolbox

This perhaps seems like a lot of attention lavished on names, particularly for a publication whose title includes the phrase "software tools." But names are an important part of a programming tool. In a well-designed hand tool, the grip gets serious attention as the primary ergonomic interface, which plays a major part in determining the effectiveness of the tool. The programmer uses names and verbal constructs to manipulate the power of the computer. If those names and constructs fit well the habits of the mind, the task is done that much more easily.

I have proposed new operators as well as new names. I believe that any Forth package should include as a standard component a complete set of double-precision operators, with quad-precision operators available as an extension for 16-bit Forths. Double precision is often needed when working with large numbers in which round-off errors must be minimized, as in accounting applications. Sometimes (in 16-bit Forths) even double precision does not offer enough range and quad precision must be used.

These are the double-precision operators I believe should be present:

```
D+ D- D* *D D/MOD D*/MOD D>
                         D= D<
2SWAP 2OVER 2DROP D2DROP 2DUP
                    D2DUP 2ROT
2, 2@ 2!
2CONSTANT 2VARIABLE
```

With these at hand, programmers can easily construct other double-precision operators that might be needed: *D/, D*/, D0=, D0>*, and so forth. I suggest D2DROP instead of 4DROP and D2DUP instead of 4DUP because the former show the intention with more clarity and less mental arithmetic.

MMSForth, published by Miller Microcomputer Services, provides a different (and complete) solution to the need for operators of higher precision. Instead of offering optional quad precision, octuple precision, and so on, MMSForth generalizes the idea of integer precision.

The Utilities option for Version 2.4 of MMSForth includes an optional extension called *N-LEN#*. The *N-LEN#* operators parallel the usual number and stack operators and use the same names except that # is included as an identifier. All the operators (*#+, #DUP, #OVER, <##, ##S, #*/MOD*, and so on) work by reference to the value of *#PREC*, which the user sets.

#PREC specifies the number of cells to be used in the arithmetic operations. Setting *#PREC* to 1 produces the normal single-precision operators, and setting *#PREC* to 2 produces the double-precision operators. But you can set it to arbitrarily high values to allow integer arithmetic of arbitrary precision.

Do not think these operations are sluggish. Test routines included with the package time the computation of Fibonacci numbers and factorials. I computed and printed the 277th Fibonacci number in 1.08 seconds and the number 46! (46 factorial) in 1.50 seconds (both on an IBM PC with the NEC V20 chip instead of the 8088). The number of (2-byte) cells of precision specified in these test routines was 50 for the Fibonacci test and 540 for the factorial test. For most uses, 540 precision seems more than ample. Allowing users to specify the number of bytes of precision they need is clearly a better solution than hand-tailoring operators of various precision.

It should be noted that Version 2.4 of MMSForth, which runs on the IBM PC and on the Radio Shack Model 4 and equivalents, is a native-mode Forth, incompatible with the normal operating systems on those machines. When MMSForth is used, it monopolizes the machine and its resources.

Names, Names, Names

One problem Forth programmers face on large projects or when working as a programming team is the number of names that are generated. John James has called this "the name explosion." Because Forth programs are best written as a collection of useful tools (short definitions with general utility), the situation is particularly acute. Compared to procedural languages, Forth systems have more names (shortness of definition) and it is more important to know them (general utility).

Forth programmers generally agree also on the importance of finding the "right" name. The criteria for rightness vary, with one major division between those who prefer playful names and those of more serious mien. Both parties agree, however, that the best names accurately and immediately convey the idea of the word's function. Both parties prefer short names to long. And both find it difficult to get precisely the right name when names must be assigned continually.

STRUCTURED PROGRAMMING
(continued from page 113)

The first problem is getting a good name. But then, once good (or even merely tolerable) names have been arrived at, they must also be remembered somehow and (in a team situation) communicated. The Forth dictionary is not really a dictionary for humans: the names are not arranged alphabetically. Some versions of Forth provide words such as *LOCATE* or *VIEW* that work reasonably well—but only if you can remember the name to begin with (does anyone have a *LOCATE* and *VIEW* that work with wildcards?) and if the documentation (comments and shadow screens) is understandable and up-to-date—or, failing that, the source code is readable.

The difficulty of using new words fluently is the same as the difficulty of speaking or writing a foreign language. Having to look up every word in a dictionary is insufferably slow. In a single-programmer shop, the programmer gradually learns his or her own language and becomes fluent in the tools he or she has created. But what is to be done in a multiperson shop, with each programmer creating several names a day? Are there regular meetings wherein the programmers present their words to each other? Do they pass around a list of their creations for others to learn and use? Do they maintain an on-line encyclopedia? When a new programmer joins the group, how is that person trained in the local language? How long does it take a programmer new to the group to become fluent in the special words that are in use?

I am in the lone-programmer category, but I would be interested in hearing how multiperson shops handle the problem of names and the problem of promulgating the general-purpose tools the programmers create. If you have found a working solution to this problem, do share it.

I also would be interested in finding out how you lone wolves keep track of your own tools. Do you sort your tools into files, each file being a toolkit for a particular purpose? Do you use precompiled overlays as toolkits? Do you keep all your words and their use in your head, or do you maintain some kind of written refer-

ence book—a dictionary, or thesaurus, or encyclopedia? Let us in on your secrets.

Fragility as Strength

Once there was a contest to define Forth in 25 words or less. My definition was "Forth is like the Tao: it is a Way, and is realized when followed. Its fragility is its strength, its simplicity is its direction." I want to talk about the seeming oxymoron in this definition: Forth's fragility being its strength.

Forth has no training wheels. If you tip over, you fall: the stack explodes, the system crashes, whatever. The design decision in creating Forth was to remove safeguards to enhance performance. For programmers accustomed to bulletproof compilers, this approach seems foolhardy. Why not have as much protection as possible?

Protection of course imposes performance penalties, but perhaps even more important is the degradation of the feedback. In high-performance machines, the flip side of responsiveness is sensitivity. The more the machine gives control to the operator, the more responsibility the operator must accept. The advantage of the operator taking control is that the operator becomes more directly connected to what is happening. This connection amplifies awareness and allows the mind and the tool to merge, providing the immediacy of feedback that more closely connects thought and action. The intimacy and control of such a connection is almost addictive, which is why people who have learned to work with such tools are so reluctant to abandon them. Racing-car drivers don't enjoy spending the day behind the wheel of a station wagon.

Robert Berkey first pointed out to me how the Forth stack, leaving the arguments nakedly exposed, also lets the programmer see what is going on. Errors surface immediately—that's the fragility—and, being discovered, are then corrected—that's the strength. Merely because Forth is fragile for the programmer does not mean that the application programs are fragile. Indeed, the very degree to which errors will out during development makes the final product that much more robust.

Fragility often accompanies flexibility. The more options the machine or language offers, the more ways it can be used against itself (fragility), but the greater diversity of needs it can address and the more quickly it can be modified (strength). A mechanical example is the Gossamer Condor, a successful human-powered aircraft. A key design decision was not to attempt to make it an unbreakable machine but to make it as simple as possible, with everything visible and accessible. Let it break, as long as it is easy to fix. That simplicity also made it flexible in the sense that it was easy to modify, and in fact the Gossamer Condor's success was based upon a process of iterative development familiar to Forth pro-

1000 0000	No more data in this 512-byte block.
0xxx xxxx	Data field consists of as many bytes as specified by the number in the low bit positions (and thus a maximum of 127 bytes). Though unimportant for decoding, it is worth noting that the data bytes contain no duplicates.
1xxx xxxx	Data field consists of a single byte (the next byte after this flag), which is to be replicated as many times as the number in the low bit positions (and thus a maximum of 127 replications).

Table 4: *Flag bit structure*

```
( Work areas )
   CREATE OUTAREA 20000 ALLOT     ( will contain uncoded image )
   OUTAREA 20000 ERASE            ( size depends on application )
   CREATE INAREA 512 ALLOT        ( work area for input blocks )

( Pointers )
   VARIABLE INBYTE        ( current byte in work area )
   VARIABLE OUTBYTE       ( current byte in output area )
:  INPOINT     ( — adr ) INBYTE @ INAREA + ;      ( next source byte )
:  OUTPOINT    ( — adr ) OUTBYTE @ OUTAREA + ;    ( next target byte )

( Flag manipulation )
( These use the encoding flags )
:  NEXTFLAG     ( — f )     INPOINT C@ ;       ( puts flag on the stack
                                               )
:  BLOCKEND?    ( f — f )   128 = ;            ( end of input block )
:  REPLICATE?   ( f — f )   128 AND ;          ( replicate next byte )
:  CHARCOUNT    ( f — n )   127 AND ;          ( # of replications or )
                                               ( # of bytes to move )

( Replication )
:  REPLICATE ( f — )                ( replicates based on count in flag )
     INBYTE INCR                      ( move past the flag )
     INPOINT C@                       ( char to replicate )
     OUTPOINT                         ( destination address )
     ROT                              ( bring flag to top )
     CHARCOUNT OVER + SWAP            ( indices = address range )
     DO DUP I C! OUTBYTE INCR LOOP    ( replicating & counting )
     DROP                             ( character replicated )
     INBYTE INCR ;                    ( to next flag location )

( Move )
:  MOVECHUNK ( f — )                ( moves # of chars specified in flag )
     CHARCOUNT DUP INBYTE INCR
     INPOINT OUTPOINT ROT CMOVE       ( move characters )
     DUP INBYTE +! OUTBYTE +! ;       ( update pointers )

( Actual decoding of block )
:  BLOCKWORK                   ( decompress the run-length encoding )
     BEGIN NEXTFLAG DUP BLOCKEND? NOT OUTBYTE @ 20000 < AND
     WHILE DUP REPLICATE? IF REPLICATE ELSE MOVECHUNK THEN
     REPEAT DROP ( flag ) ;
```

Code Example 2: *Decoding run-length encoded data*

(continued from page 115)

grammers. (The best account of the development of the Gossamer Condor and its sibling the Gossamer Albatross is the book *Gossamer Odyssey* by Morton Grosser [Boston: Houghton Mifflin, 1981]).

In this spirit, tools for developers typically lack the safeguards that programmers provide in application programs: *DROP*, for instance, doesn't check stack depth before trying to drop. Such a check would slow it down too much. The programmer is responsible for making sure that the program will always have something on the stack when *DROP* is used.

On the other hand, some safeguards don't cost much. Paul Simon pointed out in a letter that the defining word *FOR*, which appeared in this column in July 1986, could include an error check with no speed penalty.

In its final version, *FOR* expects two numbers on the stack and will crash the system if it is executed with an empty stack. This behavior, perfectly acceptable when *FOR* was mine alone, becomes arguable when *FOR* is promulgated as a tool for general use. It is easy to provide some protection. The simplest approach is to include at the beginning of *FOR*'s definition (right after *CREATE*) this phrase:

```
DEPTH 2 < ABORT" Need both array
            type and number of slots"
```

The speed of the words defined by *FOR* is unaffected by this additional check. On the whole, putting in this bit of protection seems reasonable. Moreover, as Forth is an open-architecture language, those who don't want to spend the memory space on the error message can remove the check. After all, they might reason, if *FOR* fails, it is during development, when the developer can immediately correct the condition. At run time (when the end-user is running the application program), it is not *FOR* but the words defined by *FOR* that are used, and they, of course, will work fine.

Though I tested *FOR* for suitability as a tool for other programmers (as a tiny application program), I never recognized the problem of what happens when the stack is empty. My oversight occurred because I fell into the vulgar error of testing to show that the routine works instead of viewing as a failure any test that fails to find a bug.

Glenford J. Myers observes in *The Art of Software Testing* (New York: John Wiley & Sons, 1979) that the primary difference between successful and unsuccessful test efforts is that single, critical definition: a successful test is one that finds a bug; a test that finds no bug is a failure. And Gerald Weinberg's enjoyable book *The Psychology of Computer Programming* (New York: Van Nostrand-Reinhold Co., 1971) points out that a programmer trying to find errors in his or her own work is unlikely to be successful, which is why independent testing is so important.

Run-Length Decoding

I close the column with a brief discussion of run-length decoding. One way of compressing data is run-length encoding. There may be varieties of this technique, but the one I ran across was as follows.

The data are stored in 512-byte blocks, coded in variable-length data fields. Each data field has as the first byte a flag that determines the type of field and the length of the field. The flag's bit structure determines its meaning (see Table 4, page 115).

The words in Code Example 2, page 115, decode such encoded data. In this particular application I knew that the decoded data would not exceed a length of 20,000 bytes. Each coded block is read in turn into the 512-byte input work area and is then decoded into the next available area of the output work area.

The pointers *INPOINT* and *OUTPOINT* keep track of where you are in the two areas. The flag-manipulation words take care of all flag interpretation. *REPLICATE* replicates, and *MOVECHUNK* moves a chunk of data. *BLOCKWORK* decodes the block and is used within a loop that reads each of the 512-byte input blocks into the input work area in turn.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 7.**

DR. DOBB'S CATALOG

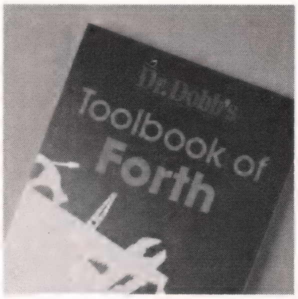
just released!

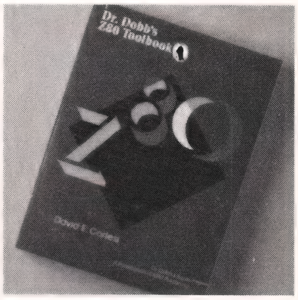
WRITING A UNIX-LIKE SHELL FOR MS-DOS

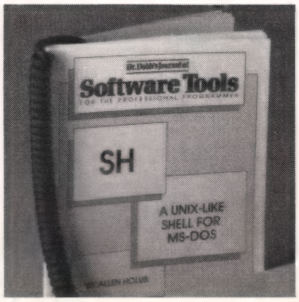
Allen Holub's new book includes an enhanced version of his popular Unix-like Shell. You'll learn how to write shells applicable to MS-DOS, as well as to most other programming environments!

ALSO INSIDE:

DR. DOBB'S COMPLETE C TOOLBOX
THE TOOLBOOK OF FORTH
Z80 TOOLBOOK
TURBO PASCAL TOOLS
DR. DOBB'S TOOLBOOK OF 68000 PROGRAMMING
TAMING MS-DOS
DR. DOBB'S BOUND VOLUMES



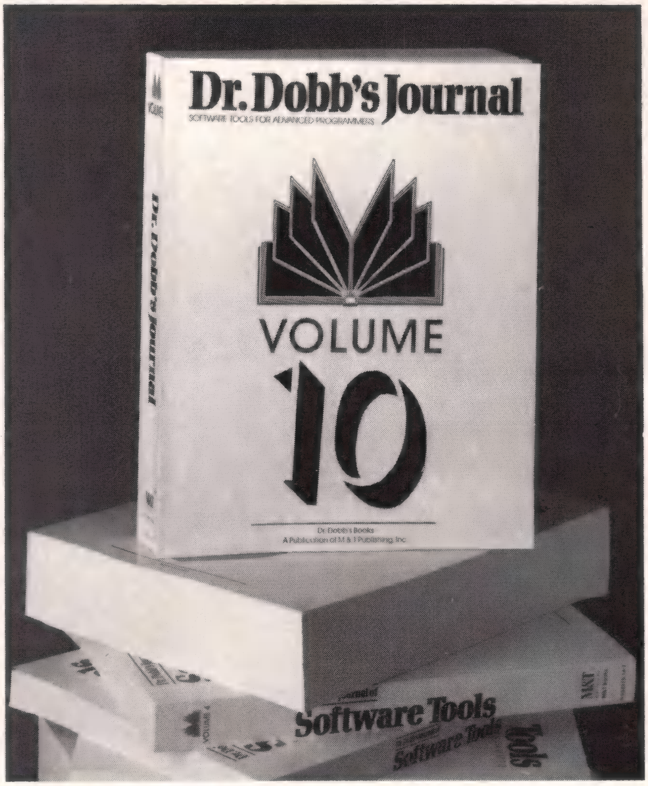




M&T BOOKS

NEW! DR. DOBB'S BOUND VOLUME 10

Bound Volume #10: The year of living dangerously. In 1985, iconoclastic DDJ beat Apple to the goal of adding more memory, a SCSI port, and a hard disk to the Macintosh. We dared to criticize the much-praised Turbo Pascal, challenged the Unix establishment with plans for a free Unix, and asked hard questions about privacy and control in the Information Age. Of course we also kept the technical level high, with the most exhaustive review ever of programmers' editors and of C compilers, and with powerful software tools in C, Modula-2, Forth, Pascal, assembly language, and Prolog.
Item #020D $35.75



Bound Volume #1: 1976 The working notes of a technological revolution. Before there was an Apple, DDJ put a programming language on the first microcomputers, and became chronicler and instrument of the microcomputer revolution.
Item #013 $30.75

Bound Volume #2: 1977 Running light without overbyte. By year two the formula was clear: serious technical questions handled with a minimum of reverence; much source code; and a commitment to tight coding.
Item #014 $30.75

Bound Volume #3: 1978 The roots of Silicon Valley growth. The S-100 bus was hashed out in DDJ's pages. Steve Wozniak and others published in DDJ code that would help build an industry.
Item #015 $30.75

Bound Volume #4: 1979 In the midst of the gold rush. Three years before IBM moved in, the neighborhood was less civilized. DDJ published a gold mine of tips, tricks, and algorithms.
Item #016 $30.75

Bound Volume #5: 1980 C and CP/M. 1980 saw an all-CP/M issue, including Gary Kildall's history of CP/M, and Ron Cain's original Small-C compiler.
Item #017 $30.75

Bound Volume #6: The First of Forth. This was the year DDJ launched its first Forth issue and Dr. Dobb's Clinic. Plus: PCNET, the Conference Tree, and 6809 Tiny BASIC.
Item #018 $30.75

Bound Volume #7: 1982 Legitimacy. DDJ observed the IBM phenomenon, reviewed MS-DOS and CP/M-86, and looked forward to fifth-generation computers.
Item #019 $35.75

Bound Volume #8: 1983 Power tools. Professional software development on a PC was getting easier; DDJ helped, with Small-C, the RED editor, and an Ada subset.
Item #020 $35.75

Bound Volume #9: 1984 Shaping things to come. In 1984 DDJ examined new programming environments: Prolog, expert systems, Modula-2, and a $49.95 Pascal. Plus Allen Holub's GREP, Unix internals, and two encryption systems.
Item #020B $35.75

SAVE! ORDER THE COMPLETE 10 VOLUME SET!

Receive all ten Bound Volumes for only $262! You save $65!
Item #020E $262

TO ORDER: RETURN THE FORM AT THE END OF THE CATALOG, OR
CALL TOLL-FREE 1-800-528-6050 EXT 4001
AND REFER TO PRODUCT ITEM NUMBER, TITLE AND DISK FORMAT

DR. DOBB'S C TOOLBOX

DR. DOBB'S TOOLBOOK OF C

Over 700 pages of C material, including articles by such C experts as Kernighan and Ritchie, Cain and Hendrix, Skjellum and Holub! The level is sophisticated and pragmatic. The most valuable part of the *Toolbook* to many will be the hundreds of pages of useful C source code, including: Jim Hendrix's famous Small-C Compiler and New Library for Small C—Also available on disk!; NEW! Hendrix's Small Mac: An Assembler for Small C and Small Tools: Programs for Text Processing—Both also available on disk!; and all of Anthony Skjellum's C Programmer's Notebook columns distilled by Tony into one thought-provoking chapter.
From M&T Publishing and Brady Communications
Dr. Dobb's Toolbook of C **Item #005** **$29.95**

SMALL-C COMPILER

Jim Hendrix's Small C Compiler is the most popular piece of software published in *Dr. Dobb's* 11-year history. Like a home-study course in compiler design, the Small-C Compiler and the *Small-C Handbook* provide everything you need but the computer for learning how compilers are constructed, and for learning C at its most fundamental level.
Small-C Compiler **Item #007** **$19.95**

SMALL-MAC:
AN ASSEMBLER FOR SMALL-C

This assembler features simplicity, portability, adaptability, and educational value. The package includes: a simplified macro facility; C language expression operators; object file visibility; descriptive error messages; and an externally defined instruction table.
You get the macro assembler, linkage editor, load-and-go loader, library manager, CPU configuration utility, and a utility to dump relocatable files. Documentation is also included.
For CP/M systems only. Please specify format.
Small-Mac **Item #012A** **$29.95**

SMALL-TOOLS:
PROGRAMS FOR TEXT PROCESSING

This package of programs performs specific, modular operations on text files, including: editing; formatting; sorting; merging; listing; printing; searching; changing; transliterating; copying; concatenating; encrypting and decrypting; replacing spaces with tabs and tabs with spaces; counting characters, words, or lines; and selecting printer fonts.
Small-Tools is supplied in source code form only. With the Small-C Compiler you can select and adapt these tools to meet your own needs. Documentation is included.
Small-Tools **Item #010A** **$29.95**

THE SMALL-C HANDBOOK

Jim Hendrix's *Small-C Handbook* is the reference book on his Small-C Compiler. In addition to describing the operation of the compiler, the book contains complete source listings to the compiler and its library of arithmetic and logical routines.
A perfect companion to the Hendrix Small-C Compiler available from *DDJ* on disk, the *Handbook* even tells you how to use the compiler to generate a new version of itself!
While both the *Handbook* and the *Toolbook* provide documentation for the Small-C Compiler, the *Handbook* contains a more detailed discussion and is available with addendum for the MS/PC-DOS version.
From M&T Publishing and Brady Communications
The Small-C Handbook **Item #006** **$17.95**
The Handbook with MS/PC-DOS Addendum
Item #006A **$22.95**

C DISK FORMATS

When ordering, please indicate MS/PC DOS or CP/M. For CP/M disks, please specify one of the follwing formats: Apple, Osborne, Kaypro, Zenith Z-100 DS/DD, 8" SS/SD. Special order formats are available for an additional $10 each.

SPECIAL PACKAGES 20% OFF

CP/M C PACKAGE

Receive this special package and save $20!
You'll get: Dr. Dobb's Toolbook for C; The Small-C Handbook; The Small-C Compiler on disk; The Small-Mac assembler on disk, with documentation; and The Small-Tools text-processing programs on disk, with documentation all for only $99.95!
Please specify format.
CP/M C Package **Item #005A** **$99.95**

MS/PC-DOS C PACKAGE

Save $20 when you order this special package.
You'll receive: Dr. Dobb's Toolbook of C; The Small-C Handbook with the MS/PC DOS Addendum; The Small-C Compiler on disk; and The Small Tools text-processing programs on disk, with manual all for only $82.95.
MS/PC-DOS C Package **Item #005B** **$82.95**

TO ORDER: RETURN THE FORM AT THE END OF THE CATALOG, OR
CALL TOLL-FREE 1-800-528-6050 EXT 4001
AND REFER TO PRODUCT ITEM NUMBER, TITLE AND DISK FORMAT

TURBO PASCAL TOOLS

TURBO ADVANTAGE: SOURCE CODE LIBRARIES FOR TURBO PASCAL

Here's the advantage you need to make your programming easier and more efficient! This library of 220 routines, complete with source code, sample programs and documentation, will save you hours developing and optimizing your programs. A few of the helpful **Turbo Advantage** files include:
* arithmetic operations * bit manipulations * check routines * data compression * differentiation and integration * Fourier analysis and synthesis * file management and hash methods * Input/Output routines * matrix processing * menu functions * MS-DOS support * linear and nonlinear optimization * screen/printer redefinition * sorting routines * spline integration, differentiation, interpolation * statistical procedures * string processing * type conversion
All procedures and functions are supplied in source code form so you can adapt them to meet your needs. Each file contains a list of relevant routines, all explained with example programs to help you understand the callings, parameters and data types in detail. Most of the programs can be immediately compiled. The detailed manual includes a short characterization of the routine, a description of the way the routine works with an explanation of the methods used, the calling sequence—including a description of the parameters and the function result, notes with advice and special explanations, and a simple example illustrating the files and variables needed.

Turbo Advantage is for MS/PC DOS systems.
Item #070 $49.95

TURBO COMPLEX ADVANTAGE: COMPLEX NUMBER ROUTINES FOR TURBO PASCAL

Working with complex numbers, vectors and matrices is easy with **Turbo Advantage Complex!** This library of over 80 procedures and routines will help you program complicated functions more easily. You'll find routines to help you:

* use digital filters, solve boundary-value problems and process very large arrays;
* carry out vector and matrix calculations with complex integers and variables;
* execute time-saving simultaneous Fourier transforms and calculations of convolution and correlation functions.

The **Turbo Complex** package includes source code and documentation. For MS-DOS systems. Some of the Turbo Complex routines are most effectively used with routines contained in the Turbo Advantage package.
Item #071 $89.95

TURBO DISPLAY ADVANTAGE: FORM GENERATOR FOR TURBO PASCAL

Now, even if you have little programming knowledge, you can design and process forms to fit your needs! The Turbo Display form generator includes a text editor, 30 time-saving Turbo Pascal procedures and functions, and a concise, informative handbook to help you take full advantage of all Turbo Display capabilities.

Turbo Display includes source code and documentation. For MS-DOS systems. Some of the Turbo Advantage routines are necessary to compile Turbo Display.
Item #072 $69.95

TO ORDER: RETURN THE FORM AT THE END OF THE CATALOG, OR CALL TOLL-FREE 1-800-528-6050 EXT 4001 AND REFER TO PRODUCT ITEM NUMBER, TITLE AND DISK FORMAT

TURBO PASCAL TOOLS

THE TURBO PASCAL TOOLBOOK

The **Turbo Pascal Toolbook** contains an extensive library of routines and sample programs to help make your programming easier and more powerful. Let Turbo Pascal experts show you how to easily integrate the routines into your own programs. Innovative sample programs demonstrate exactly how the routines are implemented.

• **Mathematical Expression Parsers** offers two routines that convert mathematical expressions into RPN-tokens.

• **The Spiderweb Database Structure** presents a new database routine that combines the best features of the B-tree, B+ and B++ trees.

• **When Turbo Isn't Enough** presents an extensive library of low-level routines that show you how to tap into the hardware and the operating system.

• **Artificial Intelligence Techniques** includes three programs that demonstrate ways to implement a user-friendly system.

• **Window Management** will help you create, sort and overlay windows.

• **A Smart Regression Model Finder** automatically searches for the best regression model to analyze a given set of data.

The routine libraries and sample programs in **The Turbo Pascal Toolbook** are also included on disk for MS-DOS systems. All source code is included.

The Turbo Pascal Toolbook	**Item #080**	**$25.95**
The Turbo Pascal Toolbook with disk	**Item#081**	**$45.95**

STAT TOOLBOX FOR TURBO PASCAL

The STAT TOOLBOX is two statistical packages in one! Whether you're a programmer looking for tools to build your own applications, or a user who wants a complete, fully functioning package, the STAT TOOLBOX will bring convenience, power and versatility to your statistics programs!
The STAT TOOLBOX is written in Turbo Pascal and includes full source code. It contains two complete packages:

A reference disk and manual
Flexible, time-saving building blocks allow you to customize statistical applications to fit your needs. You'll find:
- statistical distribution functions;
- random number generation;
- basic descriptive statistics
- parametric and non-parametric statistical testing.
- bivariate linear regression, multiple and polynomial regression
- automatic best curve selection

A demonstration disk and manual
The demonstration package includes fully functioning statistical programs, and two data management programs. In addition to these practical tools, files containing sample data will help users learn to enter, edit, maintain and manipulate data.
High resolution graphics capabilites include drawing histograms and regression lines, and plotting residuals for regression analysis. (Borland's Turbo Graphic's Toolbox is required)
For IBM PC's and compatibles. Turbo Pascal version 2.0 or later, and PC-DOS 2.0 or later are required.

Stat Toolbox	**Item #050**	**$69.95**

TO ORDER: RETURN THE FORM AT THE END OF THE CATALOG, OR CALL TOLL-FREE 1-800-528-6050 EXT 4001 AND REFER TO PRODUCT ITEM NUMBER, TITLE AND DISK FORMAT

THE DR. DOBB'S TOOLBOOK SHELF

Z80 TOOLBOOK

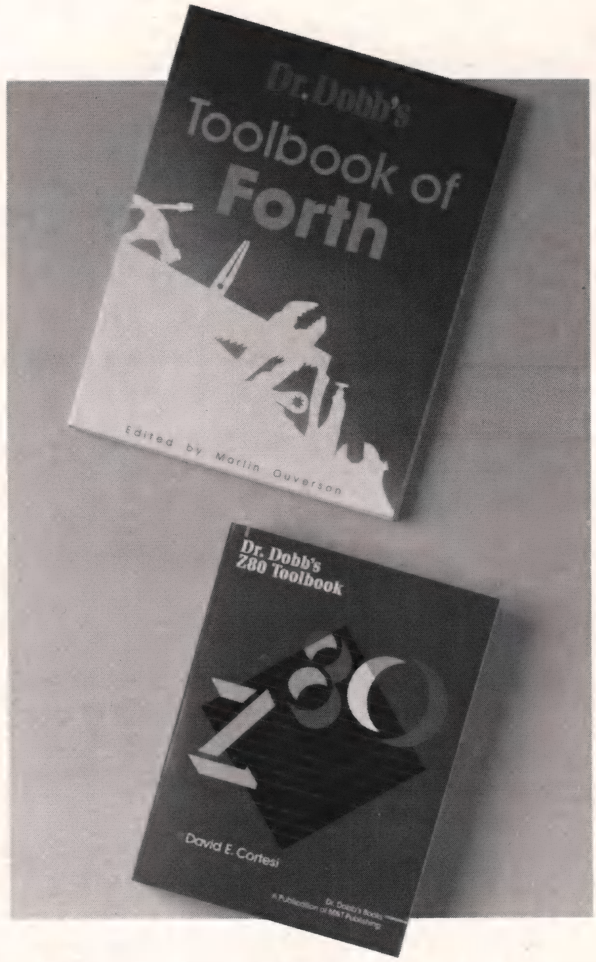
David E. Cortesi, longtime Dr. Dobb's columnist brings you:

- **a method of designing programs and coding them in assembly language.** Cortesi walks through the initial specifications, designing an algorithm and writing the code. He demonstrates the construction of several useful programs.
- **a complete, integrated toolkit of subroutines** for arithmetic, for string-handling, and for total control of the CP/M file system. They bring the ease and power of a compiler's run-time library to your assembly language work, without a compiler's size and sluggish code.
- **Every line of the toolkit's source code is there to read.**

ORDER THE Z80 SOFTWARE ON DISK!

All the software in Dr. Dobb's Z80 Toolbook—the programs plus the entire toolkit, both as source code and object modules for both CP/M 2.2 and CP/M Plus—is yours on disk! Most of the programs are included in the book, however, the disk is necessary for complete listings. A Z80 microprocessor and a Digital Research International RMAC assembler or equivalent are required.

Dr. Dobb's Z80 Toolbook **Item #022** **$25**
Dr. Dobb's Z80 Toolbook w/disk **Item #022A** **$40**
Please specify one of the following disk formats: 8" SS/SD, Apple, Osborne, or Kaypro



DR. DOBB'S TOOLBOOK OF FORTH

This comprehensive collection of useful Forth programs and tutorials contains DDJ's best Forth articles, expanded and revised along with new material. In addition, you'll glean important insights about the potential of this increasingly popular language from the many in-depth discussions of advanced Forth topics. You'll find sections on:

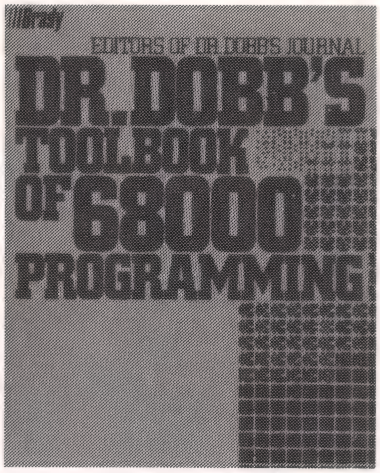
Mathematics in Forth, including "Series Expansion in Forth," "Forth Floating-Point Package," and "Signed Integer Division"
Modifications/Extensions, including "A Proposal for Strings in Forth," "Non-Deterministic Control Words," "Some Forth Coding Standards," and "Towards a More Writable Forth Syntax"
Forth Programs, including "GO in Forth," "Elements of a Forth Data-Base Design," "The Forth Sort," "SEND & RECV," "Interface for a Mouse," "Relocating Loader in Forth," "Forth Decompiler," "Screen-Oriented Editor ReVisited," "Evolution of a Video Editor," "H-19 Screen Editor," and "The Conference Tree."
Forth—the language, including "The Forth Philosophy," "Teaching Forth as a First Language," and "Forth-83 and Vocabularies"
Implementing Forth, including "Forth and the Motorola 68000," "A 68000 Forth Assembler," "A Forth Assembler for the 6502," and "Z8000 Forth." You'll also find Appendices that will help you convert fig-Forth to Forth-83, and tell you how to stay up-to-date on the latest developments and refinements of this popular language.

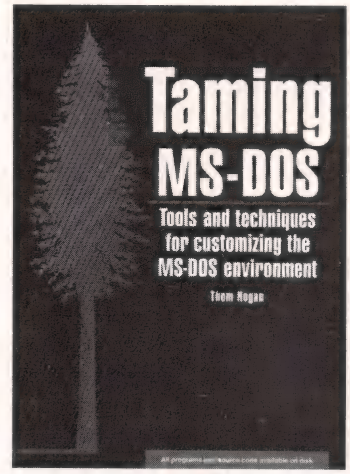
The screens in the book are also available on disk as ASCII files. Receive Dr. Dobb's Toolbook of Forth, along with the software on disk, together for only $39.95.

Dr. Dobb's Toolbook of Forth **Item #030** **$22.95**
Dr. Dobb's Toolbook of Forth w/Disk **Item #031** **$39.95**
Please specify MS/PC-DOS, Apple II, Macintosh, or CP/M. For CP/M disks, specify Osborne or 8" SS/SD.

TO ORDER: RETURN THE FORM AT THE END OF THE CATALOG, OR
CALL TOLL-FREE 1-800-528-6050 EXT 4001
AND REFER TO PRODUCT ITEM NUMBER, TITLE AND DISK FORMAT

THE DR. DOBB'S TOOLBOOK SHELF





DR. DOBB'S TOOLBOOK OF 68000 PROGRAMMING

In this complete collection of practical programming tips and techniques for the 68000 family, you'll find the best articles on 68000 programming ever published in Dr. Dobb's, along with new material from 68000 experts. You'll learn about the most important features of the 68000 microprocessor from a full description and concise discussion of its history and design. And, useful applications and examples will show you why computers using the 68000 family are easy to design, produce and upgrade. **Contents include:**

an Introduction to the 68000 Family
- 68000 Instruction Set

Development Tools
- Bringing Up the 68000: A First Step
- A 68000 Cross-Assembler

Useful 68000 Routines and Techniques
- A Simple Multitasking Kernel for Real-Time Applications
- The Worm Memory Test
- A Mandelbrot Program for the Macintosh

All programs are available on disk!

In addition, an executable version of the 68000 Cross-Assembler can be purchased along with the source code and documentation for $25. (Requires one or more disk drives and either CP/M-80, CP/M 2.2 with 64 bytes or MS-DOS with 128 bytes)

Dr. Dobb's 68000 Programming Toolbook	**Item #040**	**$29.95**
68000 Toolbook with Disk	**Item #041**	**$49.95**

Please specify one of the following disk formats: CP/M 8″, Osborne, Macintosh, Amiga, Atari 520st, MS-DOS

68000 Cross-Assembler	**Item #042**	**$25**

TAMING MS-DOS BY THOM HOGAN

Learn how to make DOS work for YOU! **Taming MS-DOS** will take you beyond the basics, picking up where your DOS manual leaves off. You'll find batch files and DOS enhancements that extend the power of DOS so you can work more accurately and efficiently. And, you'll learn how to customize DOS to fit YOUR needs, saving you time and frustration every time you use it.

Taming MS-DOS will show you how to create a memory-resident clock, rename subdirectories and change file attributes. You'll learn more about what's on your disk and how to protect it easily, how to create configurable AUTOEXEC.BAT files and understand redirection and piping. **You'll find:**

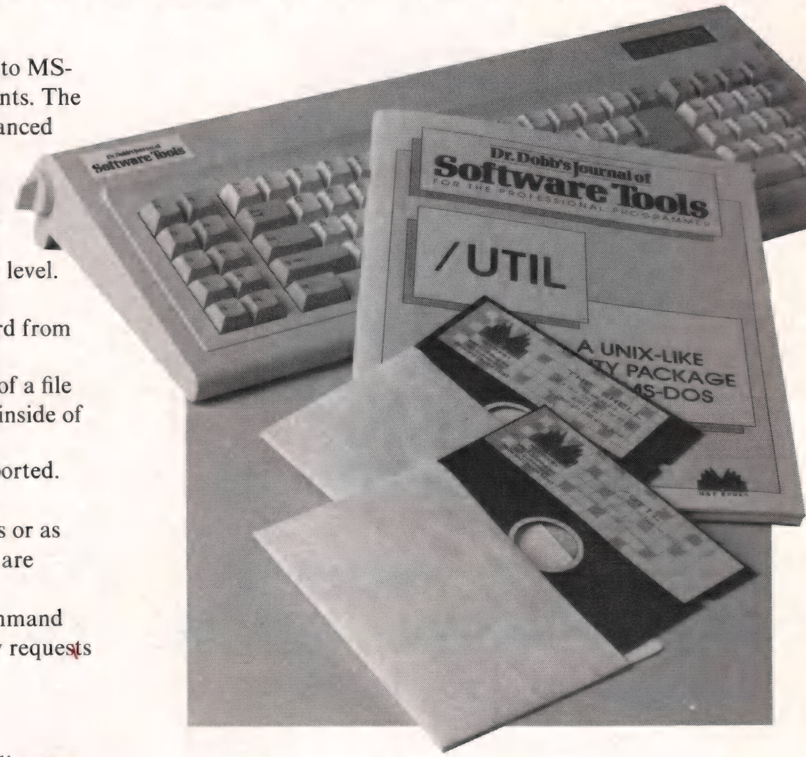
- how to customize CONFIG.SYS and use ANSI.SYS to change the appearance of DOS
- extensive batch file coverage
- example routines using redirection, including redirection operators, filters, and pipes
- DOS enhancement programs that are ready to use, with executable commands.
- Full source code, allowing you to alter programs and create a custom system.

The book includes assembly language programs that allow you to manipulate DOS at its lowest form. An assembler is not needed to enter the programs; **Taming MS-DOS** presents an alternative method for anyone with BASIC on their machine. The programs, including batch files and DOS enhancements are also available on disk along with source code.

Taming MS-DOS	**Item #060**	**$19.95**
Taming MS-DOS with disk	**Item #061**	**$34.95**

TO ORDER: RETURN THE FORM AT THE END OF THE CATALOG, OR CALL TOLL-FREE 1-800-528-6050 EXT 4001 AND REFER TO PRODUCT ITEM NUMBER, TITLE AND DISK FORMAT

WRITING A UNIX-LIKE SHELL FOR MS-DOS



This book will show you how to write shells applicable to MS-DOS as well as to most other programming environments. The book and disk include an in-depth description and enhanced version of Holub's popular Unix-like Shell, along with complete C source code. You'll find:

- how to do interpretive control flow in any C program;
- a thorough discussion of low-level DOS interfacing;
- significant examples of C programming at the system level.

The Shell's NEW supported features include:

NEW! **READ** Lets you use input from the keyboard from within the Shell script.

NEW! A new Shell variable expands the contents of a file so a program can produce text that is used inside of the Shell script.

Editing Command-line editing with the cursors is supported. The line is visible as you edit it.

Aliases Can be used to change the names of commands or as very fast, memory-resident, batch files. Nested aliases are supported.

History You can execute previous commands. The command can be edited before being executed. Imbedded history requests (Bar; !!>foo) are supported.

Redirection and Pipes <> >> >& >>& |
Pipe temporary files can be put on a RAM disk.

Unix-like Command Syntax / can be used to separate directory names (\ can now be used as well). A 2048-byte command line is supported. Command-line wild card expansion. Multiple commands on a line.

DOS-compatible prompt support
$d $t $| $h $n $q $$ $%

C-Shell Based Shell Scripts (batch files) Shell Variables are macros that can be used on the command line. Arithmetic manipulation of shell variables using the @ command are supported. The following C operators are also supported: ( ) + – * / % <= >= <> != == ! && || =

A batch file can call another batch file like a subroutine. Control is passed to the second file an then back to the first when the second is finished. Batch files can return values to the calling file using the exit and $status mechanisms.

A powerful, interpretive, programming language, based on the UNIX C Shell, is now supported, including:

if/then/else	foreach	break
while	switch/case	continue

All commands can be nested.
The shell runs on **IBM PC**'s and compatibles.

Writing a Unix-like Shell for MS-DOS
book & disk Item #163 **$39.95**

/UTIL

/Util is a collection of UNIX-like utility programs for MS-DOS. This package includes updates of the highly acclaimed Dr. Dobb's articles; Grep: a UNIX-like Generalized Regular Expression Processor, and LS and Getargs from DDJ's C Chest.

Source code is included and all programs (and most of the utility subroutines) are fully documented in a UNIX-style manual. You'll find executable versions of:

cat	echo	mv	rm
cp	grep	p	rmdir
date	ls	pause	sub
du	mkdir	printenv	chmod

/Util **Item #161** **$29.95**

ORDER FORM

ORDER NOW!

NAME ______________________________

(Please use street address, not P.O. Box)

ADDRESS ______________________________

CITY ____________ STATE ________ ZIP ________

DAY PHONE ______________________________

For disk orders, please indicate format. Refer to ad for standard format availability for each product. Special formats available for additional $10 each.

☐ MS/DOS ☐ CP/M ___ Zenith Z-100 DS/DD
☐ Macintosh ___ Kaypro ___ Osborne ___ Amiga
☐ Apple II ___ 8" SS/SD ___ Apple ___ Atari 520st

TO ORDER:

CALL TOLL FREE 800-533-4372

(Mon-Fri, 8-5 Pacific Time)

In Calif: 800-356-2002

Or, return your order and payment to:

M&T Publishing, 501 Galveston Dr., Redwood City, CA 94063

QUANTITY	ITEM #	DESCRIPTION	UNIT PRICE	TOTAL PRICE

SUB-TOTAL ________

CA residents must add applicable sale tax on merchandise total ______ % → SALES TAX ________

Shipping must be included with order. See rates below. → SHIPPING ________

TOTAL ORDER ________

☐ VISA NAME ON CARD ______________________________

☐ MASTERCARD ACCOUNT NO. ______________________________

☐ AMERICAN EXPRESS EXPIRATION DATE ______________________________

☐ CHECK (make checks payable to M&T Publishing) SIGNATURE ______________________________

In U.S. For Bound Volumes, add $3.25 per book. Add $9.25 for Special C Packages. For other books and disks, add $2.25 per item. *Outside U.S.* For Bound Volumes, add $6.25 per book surface mail. Add $18 surface mail for Special C Packages. For other books and disks, add $5.25 per item surface mail. Foreign airmail rates available on request.

For Faster Service and reduced shipping costs, Europeans may order direct from: Markt & Technik, Buchverlag, Hans-Pinsel-Strasse 2, 8013 Haar bei München. Call Germany 89-4613-221 for prices in Deutch Marks.

PROMPT DELIVERY! DEALER INQUIRY WELCOME!

DR. DOBB'S CATALOG
ORDER FORM

Please fold along fold-line and staple or tape closed.

No Postage
Necessary
If Mailed
In The
United States

BUSINESS REPLY MAIL

First Class Permit No. 790 Redwood City, CA

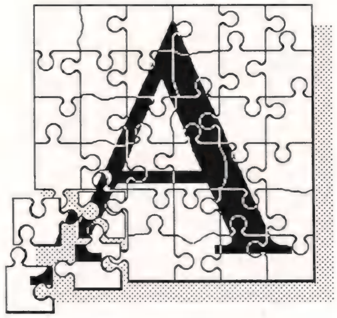
Postage Will Be Paid By Addressee

DR.DOBB'S CATALOG

501 GALVESTON DRIVE
REDWOOD CITY, CA 94063

Please fold along fold-line and staple or tape closed.

A New Project Is Born



I'd like to design a versatile, easy-to-use interpreted language, using occasional essays in this space to stimulate my own creative juices and get feedback from you. My approach to this project will be experimental, and the entire interpreter will be written in 680xx assembly language. Why? Because I love 680xx assembly language, and I like to noodle around looking for really efficient ways to do stuff. As my interpreted language comes together, I want to know what you think of it. If the ideas expressed here get your juices flowing, send me a letter. If you send me interesting enough letters, I'll include them here. I'd like this to be both an educational project for interpreter designers and a general discussion of data handling in assembly language.

Why an Interpreted Language?

The nicest thing about an interpreted language is that it can be very interactive, and if it's extensible and fast enough, it can be a real joy to work with. Forth is an example of the kind of language I'm talking about. I like Forth a lot. But the problem with Forth is that it's too weird—I find I have to think backward to use it effectively, and I'd like to be able to think in my most efficient way—forward. Thus, because I've never seen a true interpreted language that satisfies me, I want to design my own, with your feedback to help guide me.

by Nick Turner

Juggling Numbers

In this first essay I want to talk about math; specifically, numeric formats. This is intended as both an introduction to numeric representation (for those who may not have a lot of low-level practice) and as a source of inspiration for experienced assembly-language programmers. I'll start things off with a summary of some of the various numeric formats that have been used on computer systems. This will be a general description; more detailed stuff will come later on. I hope to make most of these formats available in the final interpreter.

Simple Integers

The simplest approach to computer math is to use integers. Though integer (INT) math may initially seem rather limited, a surprising amount of complex calculation can be done with integers alone. On typical computer systems, there are usually three kinds of integers: bytes, words (two bytes), and long words (two words). Sometimes you might need extra-precision integers of eight or more bytes. I propose at least two kinds of integers for my interpreter: word size (INT) and long word size (LINT). Both would be signed values, with negative numbers expressed in two's-complement form. Will I need double-long, 8-byte integers (DINT)?

Simple math with integers is straightforward. The biggest advantage of INT math is speed. Overflow and underflow are typically the most important error conditions. The biggest practical disadvantage of integer math is the inability to represent fractional values directly. Fractions can be represented by multiplying all the numbers in the system by some constant, but it requires extra time and programming. Besides, if the constant multiplier is a power of 2, you've just invented the next category: fixed-point numbers.

Fixed-Point Numbers

A typical fixed-point (FIX) representation allocates a number of bits for the integer portion of a value and an equal number of bits for the fractional portion. For example, you might use a 4-byte long word in which the high-order word is the integer and the low-order word is the fraction. Some systems use larger FIX formats with a whole long word for each portion, and a few systems have unequal distributions of bits. In such cases, it's usually the fractional portion that has fewer bits. I propose one FIX format for my interpreter (mostly for speed in calculations involving fractions). My FIX could be two long words—one for the integer and one for the fraction. The high bit of the integer portion would be reserved for the sign, and the rest would be an unsigned value. (This simplifies output of ASCII translations of the number.)

FIX has the advantage of being able to deal with fractions, but it still has the problem of limited precision, especially for small numbers. From here there are two directions in which to go. Which path a system takes depends on what the numbers will be used for. If the ability to represent really huge or miniscule values is more important than vastly precise representations, then floating point is probably best. On the other hand, if incredibly high precision is necessary, you might choose what I call extended representation.

Floating-Point Numbers

By far the most frequent choice in typical systems is a floating-point representation (FLOAT), in which the value is divided into two subvalues: the exponent and the mantissa. The exponent represents the logarithm in base 2 of a number by which the mantissa is to be multiplied to create the actual value stored. For example, if the exponent is 4 and the mantissa

is 3, then the value might be 3 times 2 to the fourth power, or 3 times 16, or 48. In actual practice, the mantissa is almost always treated as a fraction. In the above case, the exponent would be 6 and the mantissa would be 0.11 (binary), which is 3 (or 11 binary) shifted left twice. Note that the exponent really represents nothing more than the number of times the mantissa must be shifted to create the actual value. If the exponent is negative, you shift the mantissa to the right. If it's positive, you shift it left. The mantissa usually also has a sign bit, which governs the sign of the entire value.

Now here's the tricky part about floating point: most FLOAT representations nowadays have something called a "hidden 1 bit." This means that the high-order bit of the mantissa, which is always a 1 bit in a properly normalized FLOAT value, is "overlaid" by the sign bit of the mantissa or is omitted altogether. The cost of this 1-bit saving is that the missing bit must be recreated every time a calculation is done. For systems with a hardware assist, such as the MC68881 floating-point math chip, this is trivial. Another tricky point is that the exponent is usually represented as an "augmented" value—this means you must first subtract a certain number from it in order to get the actual exponent. The augment number is chosen such that an exponent of zero is represented as a bit field with only the high bit set. The result is that the exponent can be treated as a simple unsigned value, simplifying many calculations.

For my interpreter, I propose the FLOAT formats used by the MC68881 chip—specifically, the single, double, and extended representations, which I will call FLOAT1, FLOAT2, and FLOATX for my language. The reason is simple: I'd like to use the 68881 chip eventually.

A Weird Extended Hybrid

The last approach to numeric representation, and one that I've not seen used very much, is sort of a weird hybrid between floating point and fixed point. I call it extended representation (EXT), and it's the only numeric format in my proposed system that uses variable-length fields. The basic concept is simple: a number is represented in full precision as a large field of 2-byte words, with a giv-

THE RIGHT TO ASSEMBLE
(continued from page 127)

en number of words representing the integer portion (except the highest order bit, which is the sign bit) and the remainder representing the fractional part. Of course, there must also be a field somewhere that contains some clue as to where the radix point is (the radix point separates the integer from the fraction). It's also important to have a value that says how long the whole thing is.

The EXT format has certain advantages for a limited set of problems. For instance, I've always wanted to be able to compute various irrational values to an arbitrarily high precision. My EXT format can do this, but problems arise. For example, as soon as you attempt to calculate a transcendental function, you run into precision vs. time trade-offs: if you use the traditional polynomial approximation method, your polynomial factors will limit the precision of the result, which must then be chopped accordingly. On the other hand, if you use the full Taylor (or similar) series to compute the transcendental result, you may end up spending an inordinate amount of time to get the desired accuracy. I'm very interested in feedback on this issue; I have by no means reached a satisfying resolution.

Do You Want More?

If there's a good response to this essay, I'll continue the story. Future topics might include a detailed expansion on each of the numeric formats described here, with listings of working math routines and a discussion of the "housekeeping" information surrounding the number formats—how does the system know what kind of number it's dealing with and how does it keep track of all the variables? I'd also like to discuss the actual syntax and interface of the language—but first I'd like to see your blue-sky suggestions. What would your ideal interpreted language look like? Write to me care of *DDJ*.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 8.**

Version 3.0

New Features

Windows, Data Entry, Help Management, Menus, Text Editing, plus ...

SOURCE CODE

FREE

Vitamin C

It's good for your system!

The Vitamin C Difference

With **Vitamin C**, your applications come alive with windows that explode into view! Data entry windows and menus become a snap, and context sensitive pop-up help messages are nearly automatic.

With **VCScreen**, you'll save time by interactively painting windows and forms so what you see is what you get! Then, one button generates C source code ready to plug into your program and link with Vitamin C.

Easy enough for the beginner. Versatile enough for the professional. Vitamin C's **open-ended design** is full of "hooks" so you can intercept and "plug-in" special handlers to customize or add features to most routines.

Of course, Vitamin C **includes all source code FREE,** with no hidden charges. *It always has.* That means you'll have everything you need to adapt to special needs without spending hundreds of dollars more.

Windows

Create as many windows as you like with one easy function. Vitamin C automatically takes care of complicated tasks like saving and restoring the area under a window.

Options include titles, borders, colors, pop-up, pull-down, zoom-in, 4-way scrolling, scroll bars, sizes up to 32k, text file display & editing, cursor display, and more.

Unique built-in feature lets users move and resize windows during run-time via a definable key.

Access the current window by default or a specific window any time, even if it's hidden or invisible. Save and load windows on disk for more versatility!

Data Entry

Flexible dBase-like data entry and display routines feature protected, invisible, required, and scrolling fields. Picture clause formatting, full color/attribute control, selection sets, single field and full screen input, and unlimited data validation via standard and user definable routines. That means you aren't locked into one way of doing things.

Vitamin C even provides true right-to-left input of numeric fields with dynamic display of separators & currency symbols.

High Level Functions

Use our intergrated help management, multi-level menus, and text file routines, or build your own handlers using Vitamin C's basic windowing and data entry routines.

Standard help handler provides context sensitive pop-up help messages any time the program awaits key strokes. The help text file is stored on disk and indexed for quick access. So easy to use that a single function initializes & services requests by opening a window, locating, formatting, displaying, and paging through the message.

Multi-level "Macintosh" & "Lotus" style menus make user interfaces and front ends a snap. Menus can call other menus, functions, even data entry screens, quickly and easily.

Text editor windows can be opened for pop-up note pads, memo fields, or general purpose editing. Features include insert, delete, word wrap, and paragraph formatting.

VCScreen

Screen Painter/Code Generator

Just as Vitamin C's reusable functions speed your programming, VCSreen makes it even *faster and easier* by automatically generating C source code for your data entry screens!

With VCScreen's interactive screen editor, you actually draw your forms. You can define input, output and constant fields, headings, boxes, lines and even a window for the form to run in.

What you see is what you get. If you don't like the position of an object, just "pick it up" with the cursor and move it! Changing colors, attributes, copying, and deleting is just as easy.

VCScreen generates readable C source code. It declares variables with names you provide and can even generate structures.

With VCScreen choosing the right functions, parameters and sequences, and Vitamin C supplying the functions to choose from, you can stop worrying about semi-colons, matching braces, and calling conventions and concentrate on creating your application!

30 Day Money Back Guarantee

Better than a brochure. More than a demo disk. If you're not satisfied, simply return the package within 30 days and receive a full refund of the purchase price.

Vitamin C $225.00
Includes ready to use libraries, tutorial, reference manual, demo, sample, and example programs, and quick reference card. For IBM PC and compatibles. Specify Microsoft, Lattice, Computer Innovations, Aztec, Mark Williams, Wizard, DeSmet, or Datalight C compiler AND compiler version number when ordering.

Vitamin C Source ... FREE*
**Free with purchase of Vitamin C*

VCScreen $99.95
Requires Vitamin C and IBM PC/XT/AT or true compatible.

ALL ORDERS:

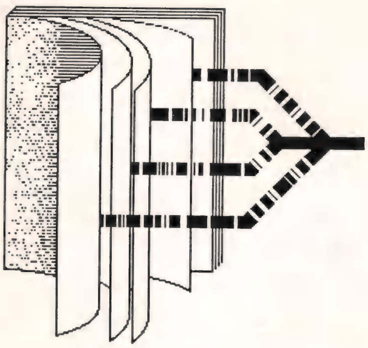
SHIPPING: $3 ground, $6 2-day air, $20 overnight, $30 overseas. Visa and Master Card accepted. All funds must be U.S. dollars drawn on a U.S. Bank. Texas residents add 6⅛% sales tax.

For Orders or More Information, Call ...

(214) 245-6090

creative
PROGRAMMING
Creative Programming Consultants, Inc.
Box 112097 Carrollton, Texas 75011

DDJ ON LINE



PROLOG and the Future of AI

The following is an excerpt from a real-time conference held by Borland International on CompuServe on July 26, 1986. A complete transcript of this three-hour on-line conference on AI and PROLOG can be found in DL6 of the Borland SIG on CompuServe (<GO BOR100> KEYWORDS:CONFERENCE).

Larry Kraft, SYSOP of Borland SIG: Our panel of featured "speakers" today includes Borland's president, Philippe Kahn; assistant professor Mark Chignell of USC, and Mike Swaine, editor-in-chief of *Dr. Dobb's Journal of Software Tools.* The first part of this conference will consist of a panel discussion centered on numerous questions that were submitted in advance. Our first panelist to speak will be Mark Chignell.

Mark: I'll start with a little information on my background in AI and PROLOG. I am an assistant professor in the Department of Industrial and Systems Engineering at the University of Southern California. I have a Ph.D. in psychology and an M.S. in industrial and systems engineering. At USC I became interested in PROLOG as a practical implementation language for AI applications in engineering. My current research is concerned with the development of human-computer interfaces in engineering design and on-line information retrieval.

Here's the first question I'm going to answer: What is artificial intelligence? People usually point to smart computer programs and say, "That's AI." In the early days of AI, 1956–1970, AI was thought of as a process of domain-independent, general-purpose reasoning. More recently, people have focused on domain-specific knowledge and the kind of heuristic reasoning that experts use. Perhaps the main unifying feature of all AI applications is the element of machine reasoning. In vision, for instance, the program is reasoning about how to update its model of the visual environment based on the sensory data. In planning, the program is reasoning about how to act on its model of the task environment so that a set of goals can be achieved and so on. (See P. McCorduck, *Machines Who Think*, for a historical introduction to the issues faced by AI.)

Philippe: Well, to me AI is what hasn't been done yet—once it's been written, it's called programs!

Mark: Question: What are "true" AI applications? Perhaps the thing that distinguishes AI from other applications is the *need* for symbolic reasoning. In statistics, for instance, it wouldn't make much sense to use an AI program to find the straight line that had the best least-squares fit to a set of data. That task is already done well by numerical algorithms. In machine chess, brute-force methods based on grinding through possible move sequences do yield fairly good results, but the problem of combinatorial explosion of search possibilities has led to an examination of how human masters perform the task and has led to attempts to incorporate their knowledge representations and heuristics into chess programs.

Philippe: Well, that is one way of typing things. . . . On the other hand, I think that five years ago people would have called a resident, beeping spelling checker AI.

Mark: I guess we have a difference of opinion here. I think it should be possible to characterize AI independently from the current status of technology. We should be moving toward a definition of intelligence that covers both humans and machines.

Question: How can you tell if a program is intelligent? In today's climate, there is a tendency to assign the label AI rather liberally. Deciding on whether a program is intelligent is a particular case of the general problem of recognizing intelligent behavior. One method for establishing the intelligence of a program is a type of Turing test. If the performances of a program and of a person on a task that requires intelligence are virtually indistinguishable, then we assume the program is intelligent.

Larry: Thank you very much, Mark. Mike, you're up next. Go ahead, please.

Mike: I'm Mike Swaine, editor-in-chief of *Dr. Dobb's Journal of Software Tools.* My background includes graduate study in both human cognition and artificial intelligence and three years reporting on AI and new technologies as a senior editor for *InfoWorld*. I am coauthor of *Fire in the Valley*, a history of the personal computer, and creator of the fictional puzzle-detective Mr. Usasi.

Question: What is declarative programming, and why would you want to use this type of programming? Declarative programming stresses static aspects of knowledge: facts about the world and rules about how the facts are connected. It concentrates on representing these facts and rules, and it deliberately submerges all procedural details. These procedural details are nothing less than the entire control structure of the program—that is, what statement gets executed next or, in more conceptual terms, how to use these static facts and rules to answer questions, solve problems, or derive new facts and rules.

PROLOG, for example, uses the model of first-order predicate logic to represent the facts and rules about some domain of knowledge—such as U.S. geography—and submerges the procedural details in an inference engine, a mechanism that automatically makes the necessary deductions from the facts and rules. To oversimplify, using PROLOG means pouring facts and rules into the system, asking questions, and letting the system derive the answers from the informa-

tion you have supplied. You declare; it deduces. To the extent that declarative programming actually submerges the procedural details, it achieves one of the goals of what is called fifth-generation language design: it allows the programmer to focus on the problem rather than on the program. In implementing a geographical database, for example, you can concentrate on facts about U.S. geography rather than on details of database design.

Question: What are the advantages and disadvantages of declarative vs. procedural programming? The choice of a declarative or a procedural approach to solving a particular problem can depend on what kind of knowledge about the problem is most accessible. If you can gather the important facts and rules about the problem domain easily, then you should consider a declarative approach. If it's easier to specify the steps or techniques for solving the problem, then you should consider a procedural approach. Another consideration is consistency vs. efficiency. A declarative, first-order, predicate-logic-based approach can be trusted to be consistent; you won't get false conclusions from true premises. But by giving up control over the way the program searches for solutions, you give up the option of fine-tuning the code for efficiency. A procedural approach lets you specify how to solve the problem efficiently but at the cost of introducing complexities that make it harder to trust the results.

Question: Are PROLOG and LISP both declarative? No language is strictly one or the other, although most programming languages are mainly procedural. LISP can be thought of as declarative, but it's a funny fit—LISP wants to be thought of as functional, and it's old enough to be humored. PROLOG was designed to be used declaratively. PROLOG probably gains in efficiency by not being purely declarative, but it pays for it in inconsistency because of extensions to the basic idea and, I think, to the way in which falsity is implemented.

Philippe: Well, LISP really manipulates functions, and I would call it procedural, just like its contemporary FORTRAN.

Mike: OK. Next question: Why is PROLOG appropriate for writing expert systems, and what systems have been written in PROLOG? Last week I watched an expert systems "knowledge engineer" being grilled by a roomful of skeptical C programmers. The C programmers all wanted to know "What do you do that can't be done in C?" The knowledge engineer had to admit that anything he did could be done in C and that in fact his company typically ported its products to C for efficiency and portability. The programmers already knew these things, but it made them feel good to hear them.

So why use PROLOG for expert systems if you'll eventually rewrite them in C? Well, that's almost like asking why use a graphics language for graphics processing. Expert systems logically include certain components that are built into PROLOG—like an inference engine; like a natural mechanism for adding to the knowledge base without rewriting the entire program. In fact, expert systems and PROLOG grew out of the same motivation: a desire to represent static knowledge in a computer program. Writing an expert system in PROLOG involves using some powerful tools. Rewriting it in C means recreating those tools. The latter may allow opportunities to optimize, but it also distracts attention from the real task.

As of today, though, PROLOG is not the language of choice for developing expert systems because of the past lack of a decent PROLOG programming environment. Of the hundreds of expert systems in nonacademic use in the U.S., nearly all were developed in some version of LISP, in a specialized expert-system-development language such as Teknowledge's S.1, or in a conventional third-generation language such as C. (One

SOFTSTRIP®
NOW OFFERS YOU SOMETHING NEVER BEFORE AVAILABLE...

CONVENTIONAL DATA HANDLING

THE SOFTSTRIP SYSTEM

A CHOICE.

Until now you were stuck with disks.

No more. Install our unique STRIPPER™ software on your personal computer today and discover the many benefits of the fastest, easiest, least expensive way to handle information.

STRIPPER lets you print – ON PAPER – your own machine readable Softstrip data strips using your dot matrix printer. The Softstrip System Reader reads that information into a computer rapidly. With STRIPPER and the reader, your PC and printer become part of the most versatile information handling system available.

With this system you can do anything you wish with any data you have in your PC – ON PAPER.

DATA ENTRY: Why use keystrokes when you can eliminate them with data strips? Whatever the document - invoices, packing slips, memos, letters, sales reports, the list is endless — simply print a data strip right on the same printed page. Now you have a document that is both human readable and machine readable. A typical document can be entered in only 15 seconds using data strips. And, it ends keystroke errors forever.

DATA DISTRIBUTION: Why copy disks? It's time consuming and expensive. Softstrip data strips will end all that. Simply photocopy as many data strips as you like and send them by mail. Data strips ignore folding, coffee stains, ink marks and, by the way, magnetic fields. And if you're using telecommunications, you can stop making the phone company rich.

DATA STORAGE AND RETRIEVAL: Why have a file of disks and a file of paper? Eliminate one with Softstrip data strips. File the data strip with the document. Better still, print the strip right on the document. Then put it in a file or binder.

Retrieval is simple. To find existing data, pull the document and its related data strip from the file. They've been stored together. Then use the reader to enter the data. No more hassle trying to match documents with the right disk — if you can find it.

DATA TRANSFER: Why bother with cables, modems and phone lines to move files between computers? A Softstrip data strip generated by an IBM PC can be read into another PC, or compatible, an Apple or even a Macintosh. If you work at home on a Macintosh, make a data strip on your printer, take it into the office and read it into your IBM PC. Simple. And we've created the utilities to let you do that easily. (See Application Notes on opposite page.)

Fascinating, isn't it? Anything you can do with disks can be done with the Softstrip data strip system — faster, easier and at lower cost — ON PAPER.

All you need is STRIPPER software at **$19.95** and the Softstrip System Reader at **$199.95.**

DDJ ON LINE
(continued from page 131)

counterexample to keep you awake nights: Lockheed is using PROLOG to develop an expert system for the Department of Defense [DoD] to analyze electronic intelligence data to determine "enemy intentions.") In Japan, PROLOG-based expert systems have been developed for (at least) medical, commercial, and engineering applications. The new PROLOG implementations coming to market may change this picture radically.

Philippe: Well, in Europe, where PROLOG was born, PROLOG has been more widely used than LISP. Furthermore, PROLOG is newer and younger, and it is just now picking up a lot of momentum.

Mark: The two other panelists classified LISP as a procedural language. As someone who has spent some time with the language, I feel some duty to defend it. LISP is really a language-development environment rather than a single language. Once you start writing functions, you can create your own language. LOGLISP is an example of a PROLOG-like language in LISP—that is, declarative versions of LISP have already been written. Object-oriented languages have also been written on top of LISP.

Mike: What does AI offer to the average programmer or user? I'll give only one of the answers; maybe others will emerge from discussion. AI is the domain of exploration of new programming techniques. When they cease to be new, they cease to be AI, but they don't cease to be useful. Also, I'd like to point out that PROLOG may be a good prototyping language for anything, not just AI applications.

I have a question. As editor-in-chief of a magazine for software developers, I am interested in the interface—in the sense of the zone of transmission—between the other two panelists' areas of expertise. I'm curious about developments in AI labs that may lead to commercial products in the future. I haven't been particularly prescient about this in the past. Having written a simple expert system in graduate school, I understood the principles. I had been following AI work closely when Teknowledge

was founded and knew the credentials of its founders; nevertheless, I did not foresee the current success of expert systems. Expert system companies are beloved of investment capitalists. Teknowledge was one of the few sales winners in a recent *San Jose Mercury News* summary of the sales slump in Silicon Valley. I'd like to do better the next time. I'd like to be able to see the next area of commercial development and practical application of laboratory developments in AI—the next Big Thing. There is a tantalizing suggestion of what that might be. . . .

Mark: That is a very good point, Mike. I think it is often hard for researchers to predict what is going to fly in the marketplace. If I were to make a bet, I would say that in the near term we may see a revolution on retrieval and utilization of information/knowledge using AI-based front ends.

Larry: OK. Philippe's turn for questions. Philippe, I believe you have some opening comments? Go ahead.

Philippe: First: My updated biography: Failed musician, mathematician, relatively artificially intelligent, self-appointed "the software industry's resident court jester"! Pops up unexpectedly anywhere.

Larry: Now, I have the questions for Philippe. What plans are there to tie PROLOG to conventional databases? Is this a growing area of AI technology?

Philippe: I don't know whether it's a growing area, but it should be very useful. The biggest problem people have with large databases, or in the AI world "knowledge bases," is reentering data. The best thing is to be able to read and write "usual" database files.

Larry: Next question: Is PROLOG a general programming language that can be used for a wide variety of programming applications or is it specifically database-oriented?

Philippe: Well, it is inference-oriented, if anything. With good extensions, PROLOG can let you do different general things, but as with any tool, you need to use the right tool for the right job. If you use a hammer when you were supposed to use a saw, you might get into trouble!

AI is the domain of exploration of new programming techniques.

Mark: You know, there are close to two billion documents on-line in the world today. This is a huge amount of information, but it's not really knowledge until you can distill the essential meaning. Perhaps the next big AI industry will be the replacement of much of the current knowledge-engineering effort with what I will call a "knowledge mining" effort, looking to translate the current backlog of electronic information into usable knowledge bases.

Philippe: There is much more in a lot of this information—things as simple as typesetting codes, tables of contents, indexes, cross-references, and so on. Millions of man-hours of editing have gone into that stuff. It is much more than dumb data, at least in many cases. You still have to interpret it, but a lot of the work has already been done. Take a book such as *Roget's Thesaurus*, for example. It divides the world into categories, and you can thus define a vector space in a given metric and talk of a much broader way to index data semantically rather than through a keyword system. But as our old friend Kipling said, "This is yet another story!"

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 9**.

The fastest C

Your search for execution speed is over. The new Microsoft® C Compiler Version 4.0 is here. With blazing performance. We've added common sub-expression elimination to our optimizer that produces code that rips through the benchmarks faster than ever before.

> "...the Microsoft performance in the benchmarks for program execution is the best of the lot overall." —William Hunt, *PC Tech Journal*, January, 1986.*

But speed isn't the only edge you get with Microsoft C. Other advantages include a variety of memory models like our new HUGE model that breaks the 64K limit on single data items. Plus our NEAR, FAR and HUGE pointers, which provide you greater flexibility. All this allows you to fine tune your program to be as small and fast as possible.

> "Excellent execution times, the fastest register sieve, and the best documentation in this review ...Microsoft Corporation has produced a tremendously useful compiler." —Christopher Skelly, *Computer Language*, February, 1986.

No more debugging hassles.
Introducing CodeView. Free.

Now, for a limited time, we'll give you an unprecedented programming tool when you buy Microsoft C, free. New Microsoft CodeView™ offers the most powerful tool yet in the war on C bugs. Forget the hex dumps. Now you can view and work with programs at any level you want. Use the program source, the disassembled object code, or

Microsoft C Compiler Version 4.00

Microsoft C Compiler
- Produces fast executables and optimized code including elimination of common sub-expressions. NEW!
- Implements register variables.
- Small, Medium and Large Memory model libraries.
- Compact and HUGE memory model libraries. NEW!
- Can mix models with NEAR, FAR and the new HUGE pointers.
- Transport source and object code between MS-DOS® and XENIX® operating systems.
- Library routines implement most of UNIX™ System V C library.
- Start-up source code to help create ROMable code. NEW!
- Full proposed ANSI C library support (except clock). NEW!
- Large number of third party support libraries available.
- Choose from three math libraries and generate in-line 8087/80287 instructions or floating point calls:
  - floating point emulator (utilizes 8087/80287 if installed).
  - 8087/80287 coprocessor support.
  - alternate math package — extra speed without an 8087/80287.
- Link your C routines with Microsoft FORTRAN (version 3.3 or higher), Microsoft Pascal (version 3.3 or higher) or Microsoft Macro Assembler.
- Microsoft Windows support and MS-DOS 3.1 networking support.
- Supports MS-DOS pathnames and input/output redirection.

Microsoft Program Maintenance Utility. NEW!
- Rebuilds your applications after your source files have changed.
- Supports macro definitions and inference rules.

Other Utilities
- Library Manager.
- Object Code Linker.
- EXE File Compression Utility.
- EXE File Header Utility.

C Benchmarks In seconds

	Microsoft C 4.0	Lattice C 3.0	Computer Innovation C 2.3	Aztec C86 3.2	Wizard C 3.0
Sieve of Eratosthenes (register)	82.9	151.4	172.3	88.0	91.9
Copy Block	86.9	231.7	199.0	123.8	189.5

Run on an IBM PC XT with 512K memory

Microsoft CodeView
Window-oriented source-level debugger. NEW!
- Watch the values of your local and global variables and expressions as you debug.
- Set conditional breakpoints on variables, expressions or memory; trace and single step.
- Watch CPU registers and flags as you execute.
- Effectively uses up to four windows.
- Debug using your original source code, the resulting disassembly or both intermingled.
- Use drop-down menus to execute CodeView commands.
- Access the on-line help to lead you through CodeView's options and settings.
- Easily debug graphics-oriented programs since program output is kept separate from debugger output.
- Keyboard or optional mouse support.
- Enter in familiar SYMDEB or DEBUG commands.

*Reprinted from *PC Tech Journal*, January 1986, copyright 1986, Ziff-Davis Publishing.

you've ever seen.

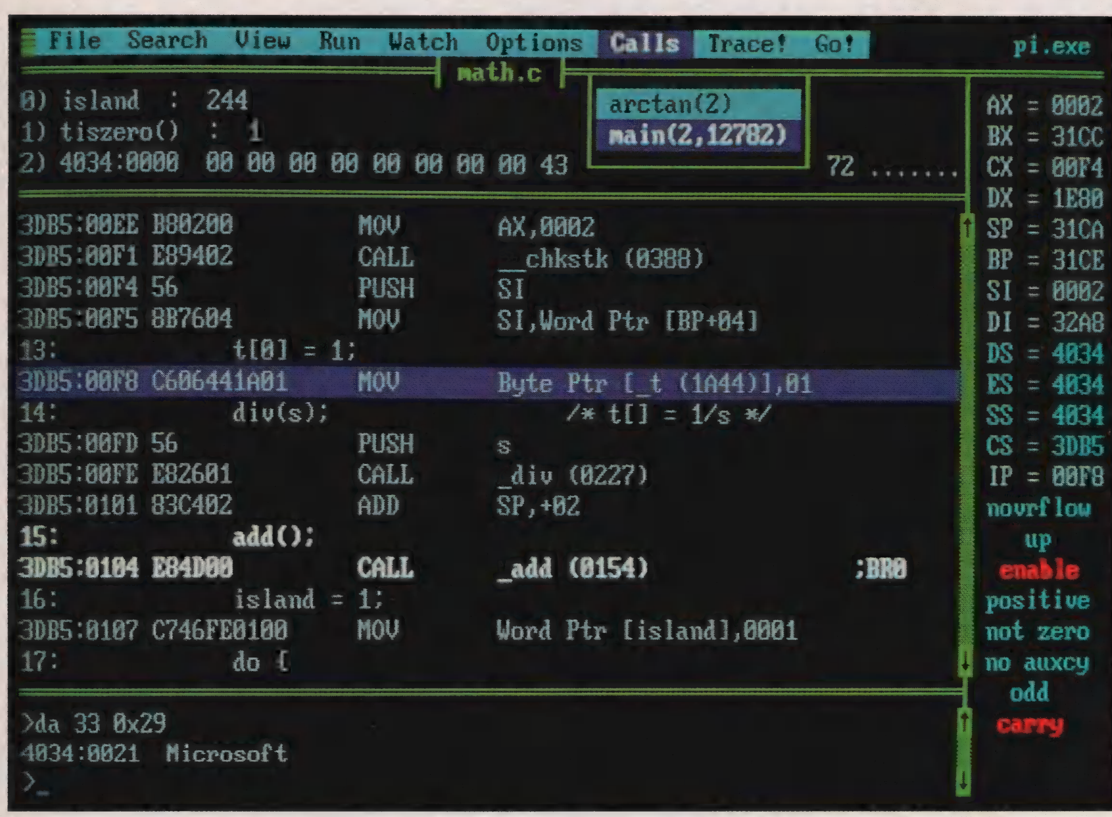
both at the same time. Open a window to view CPU registers and flags. Watch local and global variables as well. All while your program is running.

CodeView gives you complete control. Trace execution a line at a time—using source or assembly code. Or set conditional breakpoints on variables, memory or expressions. CodeView supports the familiar SYMDEB command syntax, as you'd expect. Commands are also available through drop-down menus. Combine the new window-oriented interface with our on-line help and debugging has never been easier. Or quicker.

Take the $5 CodeView tour.

You may find it hard to believe our debugger can do all we've claimed. So we're offering test drives. Five bucks will put you behind the wheel of a Microsoft C demo disk with CodeView.† See for yourself how fast debugging can get.

For more information about the CodeView demo disk, the new Microsoft C Compiler, a list of third party library support or the name of your nearest Microsoft dealer, call (800) 426-9400. In Washington State and Alaska, (206) 882-8088. In Canada call (416) 673-7638.



Microsoft® C Compiler

The High Performance Software

(continued from page 14)

indeed a principle of classical logic. As such it does not support the former invalid inference. The assumption PROLOG makes is the altogether different assumption termed the *closed-world* assumption. PROLOG automatically assumes that any postulate set (knowledge base) is complete: if it cannot be derived that *S*, then it must be that *not S*. Thus PROLOG enshrines the fallacy dubbed by Spinoza as the *argumentum ad ignorantiam:* If it can't be proven true, it's false, and if it can't be proven false, it's true. Of course it follows that a proposition that can't be proven true or false is both true and false. Indeed, with the proper repositioning of postulates (rules), PROLOG will answer "yes" and "no" to the same query even though the postulate set is consistent.

The point is that if a system is not complete (there are such complete systems—real closed fields being an example), then the assumption of completeness made by PROLOG (built into its definition of negation) is false and will lead to fallacious inferences and contradictory inferences. The justification offered for PROLOG's treatment of negation is that *not* means *not known* or *not derivable*. But this lame attempt at justification doesn't hold up. Neither the epistemic nor the apodictic concept obey DeMorgan's laws, whereas the truth-functional *not* in PROLOG does.

It is simply not safe to use *not* unless it is pinned down to a range (for example, with the use of *ON*). Otherwise the negation logic needed should be provided by the programmer. Negated sentences can be treated as units—for example, use of *not-L* instead of *not L*. The relationship between *L* and *not-L* and other negation relationships must be spelled out by the programmer. The programmer must use *(either not-L or not-K)* instead of *not(L and K)* and so forth. PROLOG never actually transforms any of the rules in the knowledge base, which means that the programmer can provide the negation logic needed.

Texts and manuals for PROLOG should be up front about PROLOG's limitations. It is not a full predicate logic in any direct sense. What PROLOG is is a negation-restricted, expanded logic of definition with marvelous recursive powers. Properly billed, the foregoing facts about PROLOG's inconsistency and radical incompleteness merely become irrelevant considerations based on a confusion about what PROLOG is supposed to be. Consider the very first problem with the introduction of a postulate declaring the transitivity of *R*. Considered as a definition, the postulate violates the cannons of definition by attempting to define *R* nonrecursively in terms of itself. PROLOG can easily handle the introduction of a transitive relation when defined recursively. The transitive closure *TR* of a relation *R* is defined:

```
TR(x y) if R(x y)
TR(x z) if R(x y) and TR(y z)
```

The general theory of definition and the theory of recursive definition can be given a rigorous syntactical formulation—it would be interesting to see an exact syntactical formulation of the extension used by PROLOG. From a computer science point of view, this would amount to giving syntactical rules to rule out non-logic-based semantic errors (logic errors in the field of partial recursive functions cannot be ruled out by syntactical rules).

The deductive-axiomatic method has been the central unifying methodology of knowledge of the Western intellectual tradition. By removing the barrier to the real-time use of the deductive-axiomatic method, PROLOG may have an impact on knowledge use and acquisition that is hard to overestimate. After all, being able to query Aristotle, or Goethe, or Einstein with an updated database does give new meaning to the expression deus ex machina.

Those who package PROLOG should have the courtesy and integrity to say what it is and what it isn't.

DDJ

Vote for your favorite feature/article.
Circle Reader Service **No. 1**.

IS GETTING THE ANSWER TO SOFTWARE PROBLEMS A BIGGER PROBLEM THAN THE PROBLEM?

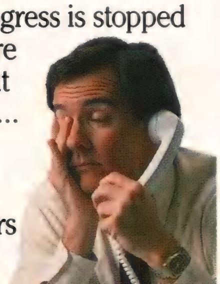
Don't stay on hold when there's help online from CompuServe® Software Forums.

The new upgraded version of your software locks up. And every time you reboot, you get stuck in the same place in the program.

You've chucked the manual, because you've done exactly what it tells you to do six times already. So you call the software company.

Now you spend half a day beating your head against a brick wall of busy signals, ranting at recorded messages, hanging around on hold. And you still don't get the solution to your problem.

Meanwhile, progress is stopped and your profits are dribbling away. But wait. There's help...



Several prominent, progressive software publishers recognize this problem, and working with CompuServe, have developed a solution—CompuServe Software Forums.

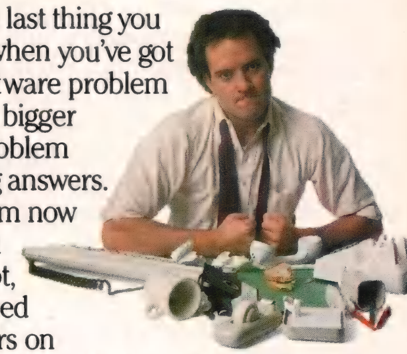
Now you can go online with experts from the companies that produced your software and get prompt, written answers to your specific problems. You can even talk with the actual software developers.

Adobe Systems®, Aldus®, Ashton-Tate®, Autodesk®, Borland International®, Creative Solutions®, Digital Research®, Living Videotext®, Lotus® Inc., Microsoft®, MicroPro®, Misosys Inc.® and Software Publishing® all have CompuServe Software Forums. And we keep adding more.

CompuServe's large subscriber base also puts you in touch with thousands of other, often more experienced, users of the same software. You'll find they can give you lots of creative ways to get the most out of your software.

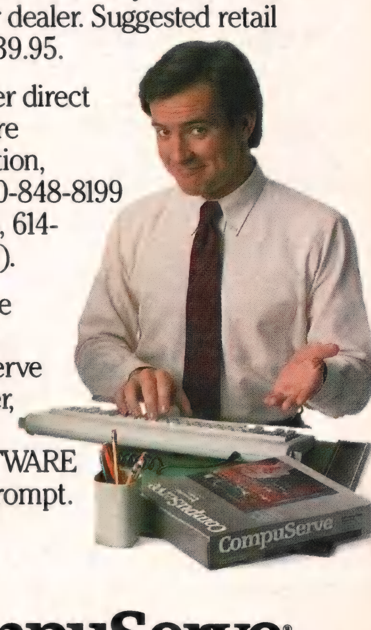
And software forums are the best way to learn about product updates, new product announcements, new ways to expand the uses of your software, and offer free uploads of your own programs.

Our online electronic magazines frequently publish software reviews. And you can find help for many other software products in our other computer-related forums for IBM®, Tandy®, Atari®, Apple®, Commodore®, TI® and others.



The last thing you need when you've got a software problem is a bigger problem getting answers. So, from now on, get prompt, informed answers on CompuServe Software Forums.

To buy your CompuServe Subscription Kit, see your nearest computer dealer. Suggested retail price is $39.95.



To order direct or for more information, call 800-848-8199 (in Ohio, 614-457-0802).

If you're already a CompuServe subscriber, just type GO SOFTWARE at any ! prompt.

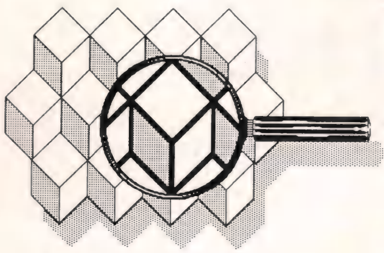
CompuServe®

Information Services, P.O. Box 20212
5000 Arlington Centre Blvd., Columbus, OH 43220

An H&R Block Company

Circle **no. 237** on reader service card.

OF INTEREST



Artificial Intelligence

ESP Frame-Engine is an expert system shell from **Expert Systems International.** Designed for customized enhancements, the shell uses frames that allow for rule-based and variable groupings as well as inheritance. Object-oriented programming is possible with several types of objects such as numeric, Boolean, text, set, and instance. The shell's open-ended architecture facilitates interfacing to Prolog-2, C, and other languages. Reader Service No. 16.
Expert Systems International
1700 Walnut St.
Philadelphia, PA 19103
(215) 735-8510

The second edition of *Experiments in Artificial Intelligence for Microcomputers* by John Krutch has been published by **Howard W. Sams & Co.** This edition contains 75 percent more material providing step-by-step procedures detailing how AI can be applied to a variety of practical activities. Programs are provided in BASIC for the Commodore 64 and 128, with instructions for converting them to other BASICs. Reader Service No. 17.
Howard W. Sams & Co.
4300 W. 62nd St.
Indianapolis, IN 46268
(800) 428-SAMS

Borland International has released an enhanced version of Turbo Prolog that provides more support for the development of large applications. The new version (1.1) has a faster compilation speed and an internal linker, with single-step compiling to executable files. It also requires less space in memory than the previous version. It costs $99.95 (free to registered owners of Version 1.0). Reader Service No. 18.
Borland International
4585 Scotts Valley Dr.
Scotts Valley, CA 95066
(408) 438-8400

Languages

CET Technology has released CET BASIC, a compiled application development language for Intel-based Unix and Xenix systems. The CET BASIC compiler is compatible with OASIS, THEOS, and UX-BASIC, and CET BASIC programs can be intermixed with programs and subroutines in other languages. Additional features include multiuser support for ISAM, direct and sequential files, terminal independence, error trapping, program chaining, and COBOL-like formatting. The CET BASIC compiler is available for $695. Reader Service No. 19.
CET Technology Inc.
5405 Garden Grove Blvd., Ste. 160
Westminster, CA 92683
(714) 895-4345

Rational Systems has released Instant C, an incremental C compiler that pares down development time by processing only those parts of a program that have been changed. The compiler incorporates a full-screen editor; source-level debugger; object code linker; source code checker; run-time checker; and support for linking Lattice C, Versions 2.0 and 3.0, and Microsoft C, Version 3.0, object code and libraries. It runs on computers with MS-DOS or Concurrent DOS and costs $495. Reader Service No. 20.
Rational Systems Inc.
P.O. Box 480
Natick, MA 01760
(617) 653-6194

Lattice now offers the SSP/PC library of more than 145 mathematical subroutines for use in scientific, engineering, and statistical computations. The subroutines can be called from Lattice C and provide PC programmers with routines similar to packages used on mainframes. Most routines yield a maximum machine accuracy of 15 significant figures. The SSP/PC library sells for $350. Reader Service No. 21.
Lattice Inc.
P.O. Box 3072
Glen Ellyn, IL 60138
(312) 858-7950

Marshal Pascal from **Marshal Language Systems** is a code-optimized ISO Pascal compiler for MS-DOS, CP/M-86, and Concurrent DOS. The compiler lets you address as much memory as your operating system allows, and it supports a variety of memory models. Marshal Pascal costs $189. Reader Service No. 22.
Marshal Language Systems
1136-P Saranap Ave.
Walnut Creek, CA 94595
(415) 947-1000

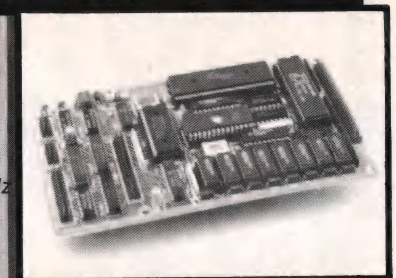
Lifeboat Associates has introduced a C++ language for micros. Advantage C++ provides extensions and enhancements, using C++ as a preprocessor to emit pure C code. C++ allows you to design your own data types and enables you to use object-oriented programming methods. Versions of Advantage C++ are available for use with Lattice C, Microsoft C, and other popular C compilers for $495. Reader Service No. 23.
Lifeboat Associates Inc.
55 South Broadway
Tarrytown, NY 10591
(914) 332-1875

The Whitewater Group has released a new object-oriented programming language that incorporates Microsoft Windows. ACTOR is a fast and powerful programming environment that uses Pascal-style syntax. DDE (Dynamic Data Exchange) has been implemented for Microsoft Windows to allow for the simultaneous transfer of information between separate programs on the same computer. ACTOR runs on the IBM PC, PC/XT, or PC/AT and sells for $495. Reader Service No. 24.
The Whitewater Group
Technology Innovation Center
906 University Pl.
Evanston, IL 60201
(312) 491-2370

Workman & Associates has released FTL Modula-2 for MS-DOS. The software provides a complete language

THE ULTIMATE IN COMPACT CP/M COMPATIBLE COMPUTERS

DSB-8100



Features:

- Hitachi 64180 CPU running at 6MHz (executes a superset of Z80 instruction set)
- 256K dynamic RAM
- 8K EPROM with boot / monitor program standard, up to 32K EPROM optional
- 1773 Floppy controller supports 40 and 80 track 5¼" and 3½" drives
- Host / target SCSI port can use DMA for all transfers
- Two RS-232 serial ports support asynchronous communications up to 38,400 baud
- Centronics type parallel printer port
- CP/M 2.2 optional
- Power requirements: +5V at 1.0A; +12V at .05A
- Size: 6-3/4" x 3-7/8"

$365.00 Quantity discounts

Compatible board with 512K RAM and 6 serial ports also available.

Davidge Corporation
P.O. Box 1896
94 Commerce Drive
Buellton, CA 93427
(805) 688-9598

Davidge

Circle **no. 353** on reader service card.

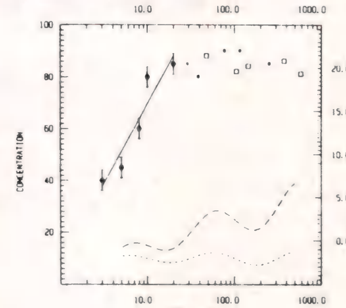
SCIENTIFIC/ENGINEERING GRAPHICS TOOLS
for the IBM PC

FORTRAN/Pascal tools: **GRAFMATIC** (screen graphics) and **PLOTMATIC** (pen plotter driver)

These packages provide 2D and 3D plotting capabilities for programmers writing in a variety of FORTRAN/Pascal environments. We support MS, R-M and IBM FORTRAN and more. PLOTMATIC supports HP or Houston Instrument plotters.

Don't want to program? Just ask for **OMNIPLOT!** Menu-driven, fully documented integrated scientific graphics. Write or call for complete information and ordering instructions.

GRAFMATIC – PLOTMATIC – OMNIPLOT [S] & [P]



Microcompatibles, 301 Prelude Drive, Silver Spring, MD 20901
(301) 593-0683

Circle **no. 286** on reader service card.

FTL MODULA-2 $49.95

The most programming power for your money

FTL MODULA-2 gives you:

- ► Full implementation of MODULA-2.
- ► Split screen, multi-file editor.
- ► Sources to the run-time modules.
- ► Complete support for the product.
- ► 8087 reals available for MSDOS.*

FTL MODULA-2 meets the standards in Niklaus Wirth's book, "Programming in MODULA-2", third edition. No other language gives you better flexibility, code design, or ease of code maintenance.

FTL MODULA-2 gives you compact, rommable code *quickly*. This is the language of the future. If you program in Pascal or C, FTL MODULA-2 will increase your output per programming hour. If you want to learn high level structured language programming, this is the best starting point.

FTL MODULA-2 was named "product of the year" by Jerry Pournelle in his column, "Computing at Chaos Manor", BYTE Magazine, April, 1986.

* 8087 support is an additionl $39.95

FTL Editor toolkit $39.95
The source code to the editor is offered as a toolkit. The editor was written in FTL MODULA-2 and this set of programs provides the novice with working sources to speed up learning. The experienced programmer will find the wealth of pre-written modules a time-saving bonanza.

SAVE $10.00
Buy both the compiler and the editor toolkit for only $79.95. That's a $10.00 savings off the regular price. **FTL MODULA-2 is available for both MSDOS and CP/M-80 systems.**

WORKMAN & ASSOCIATES
USER-FRIENDLY SOFTWARE

SEND CHECK OR MONEY ORDER TO:

Workman & Associates
1925 E. Mountain Street
Pasadena, CA 91104
(818) 791-7979 voice
(818) 791-1013 data M-F 8pm-8am
Or join us on BIX (Byte Information eXchange W.AND.A Conference.)

MasterCard and VISA welcome.
Please add $3.00 Shipping and Handling.
Calif. residents please add 6% sales tax.

Circle **no. 244** on reader service card.

(continued from page 141)

development system, including compiler, linker, integral editor, library source, assembler, and support. MS-DOS libraries include "peek and poke" throughout DOS memory, low-level DOS calls, and a debugger. The company also sells the editor's source code, which is written almost entirely in Modula-2, for $39.95. FTL Modula-2 costs $49.95. A package of FTL Modula-2 and the editor's source code costs $79.95. Reader Service No. 25.
Workman & Associates
1925 East Mountain St.
Pasadena, CA 91104
(818) 791-7979

Tools

PC/Assembler from **Computer Systems Documentation** is an interactive syntax-checking assembler for the Intel 80xx, 801xx, and 802xx and the NEC V20/30 processors. PC/Assembler is used to write assembler subprograms that can be invoked from a high-level language. It is not copy-protected and costs $99. Reader Service No. 26.
Computer Systems Documentation
P.O. Box 5478
Albuquerque, NM 87115

TurboMAGIC, a code generator for Turbo Pascal programmers, is now available from **Sophisticated Software**. TurboMAGIC includes a full-featured editor and the ability to create both pop-up and pull-down menu systems. The form image can be stored either as a typed constant or in a picture file. The software runs on the IBM, PC/XT, PC/AT, or compatible computers with 256K and is not copy-protected. It costs $99. Reader Service No. 27.
Sophisticated Software
6586 Old Shell Rd.
Mobile, AL 36608
(205) 342-7026

Expanding the IBM PC

Fort's Software has released NVRD, the Non-Volatile RAM-Disk, a software package designed to work with expanded memory hardware or with the company's Virtual Expanded Memory Manager (V-EMM). Combined with V-EMM hardware, NVRD provides the improved performance of a RAM disk with the nonvolatility of a hard disk. It runs on IBM PCs and compatibles with DOS 2.0—3.2, 192K RAM, a fixed-disk drive and fixed-disk adapter, and an EMS or V-EMM board. NVRD is available on its own for $49.95 or bundled with the V-EMM for $119.90. Reader Service No. 28.
Fort's Software
P.O. Box 396
Manhatten, KS 66502
(913) 537-2897

SOTA Technology has announced MotherCard 5.0, a plug-in card for IBM PC/XTs and compatibles that offers full AT-compatibility with all software written for the 80286, including protected mode operating systems. The on-board 1-megabyte RAM is all usable, and a DaughterCard connector is included that allows memory expansion to 16 megabytes. MotherCard 5.0 sells for $995. Reader Service No. 29.
SOTA Technology
657 N. Pastoria Blvd.
Sunnyvale, CA 94086
(408) 245-3366

An 8-megabyte memory expansion board for the IBM RT/PC is available from **Tall Tree Systems.** The JRAM-RT is a 32-bit board that makes use of the host motherboard's hardware to

(continued from page 143)

find and correct memory errors. It sells for $3,995. Reader Service No. 30.
Tall Tree Systems
1120 San Antonio Rd.
Palo Alto, CA 94303
(415) 964-1980

Micro Enhancer from **Everex Systems** is a 5-inch short card that makes EGA capabilities available for users with limited space in their IBM PC/XTs or compatibles. The board provides 640 × 350-pixel resolution graphics in 16 colors from a palette of 64 colors and is 100 percent compatible with the IBM Enhanced Graphics Adapter. To simplify using the board, Everex also supplies its EGMODE menu-driven software. Micro Enhancer costs $499. Reader Service No. 31.
Everex Systems Inc.
48431 Mimont Dr.
Fremont, CA 94538
(415) 498-1111

Personal Computer Support Group's half-slot speed-up board called the Breakthru 286 replaces the CPU of an IBM PC or PC/XT with an 80286 microprocessor faster than the one found in the 6-MHz IBM PC/AT. PCSG claims that the Breakthru 286 can beat the performance of other caching speed-up boards and that better performance can be expected in nearly all applications. The Breakthru 286 costs $595. Reader Service No. 32.
Personal Computer Support Group
11035 Harry Hines Blvd., #207
Dallas, TX 75229
(214) 351-0564

For the Mac

Datacopy Corp. has released two scanning systems—the Jet Reader and the Model 730—that produce high-resolution images containing 300 dots per square inch. Once scanned, images can be formatted for insertion into documents produced with a Macintosh desktop-publishing program. The company's MacImage software lets you control the scanner and manage, print, and view image files. The JetReader with MacImage software is priced at $2,250; the Model 730 sells for $3,250. Reader Service No. 33.
Datacopy Corp.
1215 Terra Bella Ave.
Mountain View, CA 94043
(415) 965-7900

THINK Technologies has introduced Lightspeed Pascal, a full ANSI Pascal. Lightspeed Pascal supports the Macintosh Toolbox and operating system and Apple's SANE extended numerics software. Compatible with Macintosh Pascal and Lisa Pascal, it runs on Macintosh computers with 512K RAM or more and is not copy-protected. It costs $125. Reader Service No. 34.
THINK Technologies Inc.
420 Bedford St.
Lexington, MA 02173
(617) 863-5590

Miscellaneous

Dynapro Systems has announced chronOS, a real-time multitasking operating system that lets you use standard DOS programming tools to write real-time applications. Source code is included for the console and sound generator drivers, language in-

Changing Your Address?

We'd Like to Know.

To change your address, attach your address label from the cover of the magazine to this coupon and indicate your new address below.

affix label here

Name ______________________

Address ____________________ Apt. # ____

City __________ State _____ Zip ________

Mail To:
Dr. Dobb's Journal, P.O. Box 27809, San Diego, CA 92128

(continued from page 144)

terfaces, and demo program. ChronOS runs on the IBM PC/XT or PC/AT and is tailored specifically for the iAPX86 microprocessor line. A U.S. site license costs $1,995, and a Canadian site license costs $2,495. Reader Service No. 35.

Dynapro Systems Inc.
1000 – 1200 W. 73rd Ave.
Vancouver, B.C.
Canada
(604) 263-2638

Alligator Transforms has released Prime Factor FFT, a fast Fourier transform subroutine library for the IBM PC, PC/XT, PC/AT, and compatibles equipped with an 8087 math coprocessor. The library can be called from any high-level language, and interface examples are provided in all languages. The library includes forward and inverse FFTs for single- and double-precision, floating-point, complex number sets. Users are not limited to radix2 data set sizes Prime Factor FFT sells for $159. Reader Service No. 36.

Alligator Transforms
P.O. Box 11386
Costa Mesa, CA 92627
(714) 662-0660

The Model E232-51 is a new in-circuit emulator from **Signum Systems** that provides real-time, transparant emulation for the 8031, 8051, and 8751 microcontollers. Connected to an IBM PC via the RS-232 interface, Model E232-51 features complete debugging facilities. Along with a command-driven user interface, the emulator provides users with windowing software and mouse support for controlling and monitoring program execution. Model E232-51 with 64K of overlay program memory is priced at $3,195. Reader Service No. 37.

Signum Systems
1820 14th St., Ste. 203
Santa Monica, CA 90404
(213) 450-6096

Lang-Allan has announced Version 2.00 of Bluestreak Plus, its communication package for the IBM PC, PC/XT, PC/AT, and compatibles. Bluestreak Plus is a full-featured software package that combines PC-to-PC communication and PC-to-mainframe communication with an open-architecture format for customization and modification at any level. The programming interface allows you to develop applications in many popular languages such as C, assembly language, Turbo Pascal, and dBASE. Bluestreak Plus 2.00 sells for $89. Reader Service No. 38.

Lang-Allan Inc.
2457 Aloma Ave., Ste. B
Winter Park, FL 32792
(305) 677-1539

Full-function simulation models of the Motorola MC68000 and MC68010 miroprocessors are available from **Quadtree** along with an extensive library of models of associated peripherals and an optional graphic microprocessor development system. The new models are part of Quadtree's Designer's Choice library, a library of software simulation models of standard, off-the-shelf digital devices. Call for prices. Reader Service No. 39.

Quadtree Software Corp.
1170 Rt. 22 East
Bridgewater, NJ 08807
(201) 725-2272

A 3½-hour C programming course is


C++

(continued from page 146)

available on video cassette from **Berkeley Decision/Systems**. The course is designed for expert programmers who want to learn C or people who are familiar with the language and want to learn more about it. The course includes a manual and more than 40 complete C programs to demonstrate the concepts presented in the video. *A Programmer's Introduction to C* is available for $400 in VHS or Beta formats. Reader Service No. 40.
Berkeley Decision/Systems
150 Belvedere Terr.
Santa Cruz, CA 95062
(408) 458-0500

Zoom Telephonics is now shipping a 2,400-baud version of its Zoom/Modem PC 1200. The new version has such additional features as demon dialing, audio input and output ports, and a high-speed 16450 UART for assured compatibility with IBM PC/ATs and compatibles. The Zoom/Modem PC 1200 can be upgraded to 2,400 baud with a plug-in board that costs $249. The Zoom/Modem PC 2400 ST sells for $499, and an XL version with more features costs $50 more. Reader Service No. 41.
Zoom Telephonics Inc.
207 South St.
Boston, MA 02111
(617) 423-1072

Access Associates has introduced Alegra, a memory expansion unit designed to add 512K of external memory to the Commodore Amiga. Alegra has a small footprint and allows for future expansion of up to 2 megabytes by replacing memory and configuration devices. The unit supports the auto-configuration architecture of the Amiga, and power is supplied by the computer at the expansion connector. Alegra sells for $379. Reader Service No. 42.
Access Associates
491 Aldo Ave.
Santa Clara, CA 95054
(408) 727-8520

Digitronix has released a low-cost Turbo upgrade kit called Veloz that brings IBM PC/XTs and compatibles up to the speed of a PC/AT. Veloz offers 100 percent compatibility with all major software packages and can be run with either the 8088-2 or V20 with no need to power down or replace the CPU. The price is $98. Reader Service No. 43.
Digitronix
2135 Junction Ave.
Mountain View, CA 94043
(415) 964-5103

PAX from **Baker & Rabinowitz** is a real-time multitasking executive for the IBM PC. It runs in concert with MS-DOS and requires no licensing or incorporation fees. The system kernel supports up to 32 concurrent tasks and is fully preemptive. The package is priced at $149.95. Reader Service No. 44.
Baker & Rabinowitz Inc.
3869 Kilbourne Ave.
Cincinnati, OH 45209
(513) 871-0886

CodeWorks is a new magazine for

people interested in BASIC programming under MS-DOS, TRS-DOS, or CP/M. Each annual subscription covers all six issues of the calendar year and costs $24.95. Interested readers can write for a free sample issue. Reader Service No. 45.
CodeWorks
Sample Copy Offer
3838 South Warner
Tacoma, WA 98409

A system modeling tool called Performance Analysis Tool Box is available from **Computer Technology Associates.** The package simulates a variety of centralized or distributed computer architectures, allowing designers to investigate broad ranges of use patterns in a hypothetical system. It's $10,000 for the IBM PC. Reader Service No. 46.
Computer Technology Associates
7927 Jones Branch Dr., #600W
McLean, VA 22102

Microsoft Corp. has announced the availability of extensions to the MS-DOS operating system that support the use of CD ROM disk drives with personal computers. The MS-DOS CD ROM extension consists of two software modules—a hardware-independent program and an installable device driver that must be customized by each manufacturer to work with its own hardware. Microsoft will supply the hardware-independent program—the DOS extension that will handle the much higher capacity of the CD ROMs—plus a sample device driver and documentation.

With the new software, PCs running DOS 3.1 or 3.2 can read data from any CD ROM disk that is compatible with the High Sierra Group file format proposed at the National Computer Conference in May 1986. Microsoft will license these extensions directly to CD ROM drive manufacturers, and they are available only on an OEM basis. Reader Service No. 47.
Microsoft Corp.
16011 N.E. 36th Way
P.O. Box 97017
Redmond, WA 98073-9717
(206) 882-8080

Practical Peripherals is now offering a stand-alone 1,200-bps modem, the Practical Modem 1200 SA. It is fully Hayes-compatible, includes auto-dial/auto-answer capabilities, supports virtually all communications software, and includes an upgrade path for a programmable enhancement card. The suggested retail price of the modem is $239. Reader Service No. 48.
Practical Peripherals
31245 La Baya Dr.
Westlake Village, CA 91362
(818) 991-8200

DogStar Software has announced subVol, a disk subvolume manager that brings PC-DOS- and ProDOS-like subdirectories to Apple Pascal DOS. SubVol works on any Pascal formatted disk device and allows hard-disk users to format directly and install a complete set of subvolumes with Apple Pascal.

The program works by attaching virtual disk drivers to the unused disk units in an Apple Pascal system. The virtual disk drivers cause a portion of any real disk to behave as though it were a volume in itself, including a subdirectory plus any files in that subvolume. The product runs on Apple II computers with Apple Pascal (Version 1.1, 1.2, or 1.3). The price is $34; with source code it costs $75. Reader Service No. 49.
dogStar Software
P.O. Box 302
Bloomington, IN 47402
(812) 333-5616

Instant Replay by **Nostradamus** is a demonstration development toolkit that generates tutorials, demos, presentations, menu systems, and timed keyboard macros. It is not copy-protected and runs on the IBM PC, PC/XT, and PC/AT. The product is priced at $89.95. Reader Service No. 50.
Nostradamus
5320 South 900 E, Ste. 110
Salt Lake City, UT 84117
(801) 261-0769

DDJ

ADVERTISER INDEX

Reader Service No.	Advertiser	Page No.
92	Addison Wesley	63
350	Aldebaran Laboratories	128
219	Allen Emerson & Franklin	89
321	Alpha Computer Service	73
360	Aspen Scientific	25
258	Austin Code Works	93
115	Barrington Systems, Inc.	4-5
182	BC Associates	99
267	Beacon Street Software, Inc.	86
202	Berkeley Decision/Systems	91
159	Blaise Computing	13
217	Blaise Computing	79
263	Block Island Tech	90
161	Borland International	C4
212	Burton Systems Software	14
235	C Software Toolset	86
181	C Users Group	102
*	C Ware	69
307	Cauzin Systems	132-133
*	Cauzin Systems	61
343	Circuit Research Corporation	145
122	Compu View	21
237	Compuserve	139
348	Creative Computer Software	83
*	Creative Programming	129
*	CSSL, Inc.	70
268	Custom Software Systems	145
*	DDJ Catalog	117-124
203	Datalight	9
353	Davidge Corporation	142
258	Desktop A.I.	87
89	Ecosoft, Inc.	149
90	Edward K. Rose	87
173	Entelekon	67
93	Fair-Com	65
340	Farbware	91
*	Gimpel Software	103
311	GMX	151
291	Gold Hill Computers, Inc.	1
97	Greenleaf Software	60
351	Guidelines Software	48
132	Harvard Softworks	53
233	Hawaiin Village Computers	96
780	Hersey Micro Consulting	58
293	IMSI	134
327	Integral Quality, Inc.	145
294	Kurtzberg Computer Systems	107
275	Kydor Computer Systems	93
285	Laboratory Microsystems, Inc.	95
266	Language Processors, Inc.	138
181	Lattics, Inc.	146
346	Levco	127
118	Lifeboat	29
359	Lifeboat	147
257	Logitech, Inc.	150
135	Lugaru	135
336	Magus Inc.	135
108	Manx Software Systems	7
317	Marshall Language Systems	45
285	MDS, Inc.	125
352	Metamax	83
358	MetaCom Co.	44
300	Micro Way	62
286	Microcompatibles	142
*	Micromint	54
154	Microport Systems, Inc.	27
105	Microprocessors Unlimited	102
*	Microsoft	136-137
156	Mix Software	38-39
249	Mortice Kern Systems, Inc.	131
309	Microsoft Associates	72
220	Nantucket Corporation	84
331	Nirvonics	111
243	Norton Utilities (The)	116
251	Nostradamus	2
227	Oakland Group, Inc.	47
342	Oakland Group, Inc.	26
254	Oasys	143
357	Oregon Software	C3
214	Periscope Co. Inc.	74
343	PharLap	111
239	PMI	95
229	Port - A - Soft	74
129	Programmers Connection	75
129	Programmers Connection	76-77
129	Programmers Connection	71
334	Programmer's Paradise	33
174	Programmer's Shop	43
133	Programmer's Shop	50-51
301-306, 337	Programmer's Shop	109
355	Quelo	68
206	Raima Corporation	105
145	Rational Systems	55
312	Royal American Technologies	148
*	SAS Institute	46
168	Sapiens Software	82
210	Scientific Endeavors	89
85	Semi-Disk Systems	78
78	SLR Systems	44
345	Soft Cap, Inc.	90
113	Softcraft Inc.	C2
259	Softfocus	144
361	Software Factory	114
314	Software Garden Inc.	59
347	Software Masters	125
170	Software Security, Inc.	20
148	Solution Systems	15
142	Solution Systems	101
152	Solution Systems	114
354	Sophisticated Circuits	95
287	Stonybrook Software	80
345	T.O.C Business Solutions	102
175	Tom Rettig Association	125
344	True Basic	142
230	TSF	81
207	Turbo Power Software	56
119	Turbo Tech Report	57
332	Unify Corporation	19
316	Upland Software	108
157	Vermont Creative Software	32
112	Wendin	11
116	Wizard Systems	106
244	Workman & Associates	142
225	Xenosoft	68

*This advertiser prefers to be contacted directly: see ad for phone number.

ADVERTISING SALES OFFICES

Midwest
Michele Beaty (317) 875-8093

Southeast
Gary George (404) 897-1923

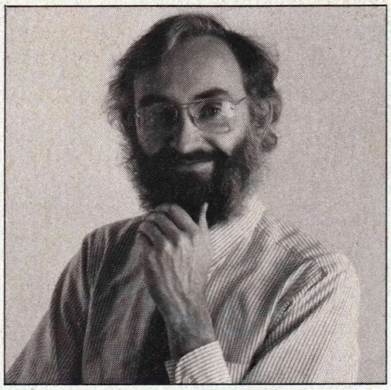
Northeast
Cynthia Zuck (718) 499-9333

Northern California/Northwest
Lisa Boudreau (415) 366-3600

Southern California/AZ/NM/TX
Michael Wiener (415) 366-3600

Advertising Director
Robert Horton (415) 366-3600

SWAINE'S FLAMES



Yes, I was kidding about BASIC in November. BASIC, on the other hand, seems to be getting serious, with such features as advanced structure constructs, new data types, labels for branching, internal and external subroutines, multiline and decision-making functions, recursion, libraries, modules, better file I/O, low-level and DOS access, sophisticated string manipulation, and advances in portability and standardization. And now that Borland is bringing out a Turbo BASIC, competition may hasten the maturation process.

Granted that the Turbo Pascal-Turbo BASIC-QuickBASIC type of rapid-interaction compiler is not a reasonable substitute for C and assembly language in a major software development task, it does seem to be finding a place in certain kinds and stages of development. The fact that both Borland and Microsoft have interactive C compilers in the works suggests that they think so—it's hard to believe that they expect to sell C to Sunday-afternoon programmers. But it's more significant that there seems to be a growing interest in enlarging the programmer's toolkit, offering more options in development environments, and offering tools that make the programmer more productive and efficient.

Good programming is often artful. Any engineering discipline is like art in that both bring new things into the world and rely on skill and serendipity and unlike art in being more concerned with result than with process. In fact, one goal of engineering is to refine processes, automating portions in order to free the engineer to be artful at another level. Does software development do this?

If not, it should. I recently reread Robert Heinlein's *The Door into Summer* and was struck by how convincingly and appealingly Heinlein portrayed the engineer as an artist whose very art provides the means to improve his brush. If software development really is an engineering discipline, it needs more Waldoes and Drafting Dans, more tools to automate the repetitive processes.

John Backus does not believe that programming is an engineering discipline yet, arguing that it is still too much an art. (See "From Function Level Semantics to Program Transformation and Optimization," *Lecture Notes in Computer Science* 185 [1982].) Solving the software crisis, he maintains, requires that it become an engineering discipline.

Parallel processing is the wave of the future, right?

That was the gist of Gordon Bell's keynote address at the Fall Joint Computer Conference in Dallas in November. The promise of parallelism may have been overstated in some areas. Two former advocates of parallel design for database machines, Haran Boral of the Israel Institute of Technology and David DeWitt of the University of Wisconsin at Madison, have come to believe that we should not build database machines that attempt to maximize throughput via massive parallelism. The problem, they contend, is that I/O bandwidth per gigabyte of storage is decreasing rapidly and that it is the I/O bandwidth issue that will be the bottleneck in database machines. "Unless mechanisms for increasing the bandwidth of mass storage devices are found, highly parallel database machines are doomed to extinction," they conclude in "Database Machines: An Idea Whose Time has Passed?"(*Database Machines*, H. O. Leilich and M. Missikoff, eds. [Berlin: Springer-Verlag, 1983]).

Those of you who live in California may have discerned a bit of chauvinism in Massachusetts' claiming to be the Software Capital of the country. Those of you who do not live in California probably detected even more chauvinism in the recent passage of a ballot proposition making English California's state language. No doubt the California legislature will soon be turning arroyos and mesas into gulches and hills just as the French, who invented chauvinism, have for years been kicking Americanisms out of *la langue Française*.

My cousin Corbett is cursing what he calls "egotistical linguistic chauvinism." He had worked hard on an alternative proposition that he thinks is much more appropriate because it actually addresses a real problem. Unfortunately, it did not get on the ballot in 1986, but Corbett plans to be more successful in 1988 and is starting to organize now. His plan is to replace English with C as California's language, and he invites your participation. C, he points out, is capable of expressing anything anyone could ever need to express, and the time has come for fanatical supporters of minority languages to put aside their pride, accept the inevitable, and end the Babel of incommensurable dialects once and for all.

"*Des egos, et encore des egos.*"—Ed Faber, in *Silicon Valley*, the French edition of *Fire in the Valley* by Paul Freiberger and me.

Michael Swaine

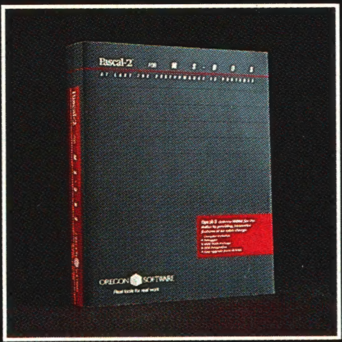
Michael Swaine
editor-in-chief

TALK OF THE TOWN

One language supports this community. That language is Pascal-2, now on the PC and producing the fastest, most compact code available. For the professional programmer, imagine what you can do with this power: ■ Cut execution time by 20% to 200% ■ Transport MS-DOS programs to VAX, PDP-11, and 68000 machines with only minor adjustments ■ Cut executable program size by up to 50% ■ Use all of DOS-addressable memory through efficient large-memory model ■ Speed error correction and save development turn-around time with sophisticated error checking and reporting ■ Find and fix logical errors with the interactive source-level debugger ■ Access DOS services and network files ■ Call Microsoft FORTRAN, C, Pascal, and assembler ■ Upgrade from TURBO Pascal with compatible strings, equivalent procedures and access to TURBO graphics.

Pascal-2™

FOR MS-DOS



Plus!

■ *Intel CEL87 mathematical library for scientific computing* ■ *A special interface between Pascal-2 and the programmable BRIEF text editor (editor optional).*
■ *Certified ISO standard Level 1.*

Dramatically improve your productivity and introduce your PC software to the VAX next door.

Call or write OREGON SOFTWARE, INC.
6915 SW Macadam Avenue,
Portland, OR 97219 (800) 367-2202
TWX: 910-464-4779 FAX: (503) 245-8449

OREGON SOFTWARE

Real tools for real work

AT LAST THE PERFORMANCE IS PORTABLE

The following are trademarks: Oregon Software, Pascal-2, Oregon Software, Inc.; IBM, PC-AT, PC-DOS International Business Machines Corporation; Intel, Intel Corporation; MS, Microsoft Corp.; TURBO Pascal, Borland International, Inc.; BRIEF, UnderWare Corp.; PDP, VAX, Digital Equipment Corp.

Circle **no. 357** on reader service card.

Why more than half a million people are using Turbo Pascal

Because Turbo Pascal is faster than any other Pascal compiler—it's become the industry standard. And because Turbo Pascal is backed by a complete range of "tool boxes" that give you most of the programming tools you'll ever need.

- Turbo Pascal® 3.0 with 8087 and BCD
- Turbo Database Toolbox™
- Turbo Graphix Toolbox®
- Turbo Tutor® 2.0
- Turbo Editor Toolbox™
- Turbo GameWorks®

The Turbo Pascal family is never static, but is continuously expanding, with new products like Turbo Editor Toolbox and Turbo Gameworks.

> "Language deal of the century . . . Turbo Pascal
> *Jeff Duntemann, PC Magazine*
>
> Turbo Pascal has got to be the best value in languages on the market today
> *Jerry Pournelle, BYTE Magazine*
>
> This compiler, produced by Borland International, is one of the best programming tools presently available for the PC
> *Michael Covington, PC Tech Journal*"

The secret of software success is not merely low price, but top quality allied with complete documentation, like our 400-page reference manual.

All of which are some of the reasons why Turbo Pascal is clearly the leader, and the recipient of awards like *PC Week's* "Product of the Year" and *PC Magazine's* "Award for Technical Excellence."

NEW! TNT* A DYNAMITE DEAL!
*Turbo 'N Tutor

Turbo Pascal *and* Turbo Tutor for only $125.00. You save nearly $15.00, and have everything you need to get up and go with Turbo Pascal!

A whole family of tools

Success breeds success, so the Turbo Pascal family has flourished. Your choices now include:

☐ **Turbo Pascal 3.0** combines the fastest Pascal compiler with an integrated development environment, *and* 8087 math co-processor support *and* Binary Coded Decimals.

☐ **Turbo Database Toolbox** a perfect complement to Turbo Pascal. It includes a complete library of Pascal procedures that allows you to search and sort data and build powerful database applications.

☐ **Turbo Graphix Toolbox** includes a library of graphics routines for Turbo Pascal programs. Lets even beginning programmers create high-resolution graphics with an IBM,® Hercules,™ or compatible graphics adapter. Does complex business graphics, easy windowing, and stores screen images to memory.

☐ **Turbo Tutor 2.0** teaches you step-by-step how to use Turbo Pascal, with commented source code for all program examples on diskette.

An offer you can't refuse

You can save $119.75 when you choose the Turbo Jumbo Pack. Six different Turbo Pascal products for only $299.95.

You get Turbo Pascal 3.0 *and* Turbo Editor Toolbox *and* Turbo Tutor 2.0 *and* Turbo Graphix Toolbox *and* Turbo GameWorks *and* Turbo Database Toolbox!

So act now—rush to your dealer or call us today!

NEW! Amazing value! Turbo Editor Toolbox includes MicroStar, a full-blown editor that also does windows! You get ready-to-compile source code and a 200-page manual that tells you how to integrate the editor procedures and functions into your programs. Also includes MicroStar, a complete editor with full windowing capabilities. (You could pay $100.00 or more for a program like MicroStar, but you get it free as part of our Turbo Editor Toolbox.) You can also use Turbo Editor (which of course integrates with Turbo Lightning®) to build your own word processor!

NEW! Turbo GameWorks gives you the games you can play, write, rewrite, bend and amend! Turbo GameWorks reveals the secrets and the strategies of game design. You're given source code, a 200-page manual, and the insight needed to write and customize your own irresistible games.

Also includes ready-to-play Chess, Bridge, and Go-Moku—an ancient Japanese game that can divert you from reality for hours on end.

YES! I want the best!

To order by phone, or for a dealer nearest you,
call (800) 255-8008
in CA call (800) 742-1133

Copies	Product	Price	Totals
___	Turbo Pascal 3.0 w/8087 & BCD**	$ 99.95	$ ______
___	Turbo Pascal for CP/M-80	69.95	$ ______
___	Turbo Database Toolbox	69.95	$ ______
___	Turbo Graphix Toolbox*	69.95	$ ______
___	Turbo Tutor 2.0	39.95	$ ______
___	Turbo Editor Toolbox*	69.95	$ ______
___	Turbo GameWorks*	69.95	$ ______
___	Turbo Pascal & Turbo Tutor	125.00	$ ______
___	Turbo Jumbo Pack*	299.95	$ ______
	Outside USA add $10 per copy CA and MA res. add sales tax		$ ______
	Amount enclosed		$ ______

Prices include shipping to all US cities.

Carefully describe your computer system:

Mine is: __ 8-bit __ 16-bit

I use: __ PC-DOS __ MS-DOS __ CP/M-80 __ CP/M-86

My computer's name and model is: ______________

The disk size I use is: ☐ 3½" ☐ 5¼" ☐ 8"

Payment: VISA MC Money Order Check

Credit card expiration date ___/___

Card # |_|_|_|_|_|_|_|_|_|

TP11

Name: ______________

Shipping Address: ______________

City: ______________

State: ______________ Zip: ______________

Telephone: ______________

CODs and purchase orders WILL NOT be accepted by Borland. Outside USA make payment by bank draft, payable in US dollars drawn on a US bank.

NOT COPY PROTECTED
60-DAY MONEY-BACK GUARANTEE

If within 60 days of purchase this product does not perform in accordance with our claims, call our customer service department and we will gladly arrange a refund.

All prices are suggested list prices and are subject to change without notice.

Minimum System Requirements:
Turbo GameWorks, Turbo Graphix Toolbox, Turbo Tutor, & Turbo Editor Toolbox—192K. All other products, 128K.
*IBM PC, PCjr, AT, XT, and true compatibles.
**16-bit only.

BORLAND INTERNATIONAL
4585 SCOTTS VALLEY DRIVE
SCOTTS VALLEY, CA 95066
(408) 438-8400 TELEX: 172373

Vive la différence

Circle **no. 161** on reader service card.